国外计算机科学教材系列 ①

中文书名	英文书名	作者	译审者	版别
离散数学（第六版）	Discrete Mathematics, 6E	Richard Johnsonbaugh	石纯一	中、英
数据结构与算法分析（C++版）（第二版）	A Practice Introduction to Data Structures and Algorithm Analysis, 2E	Clifford A. Shaffer	张 铭	中、英
数据结构与问题求解（Java版）（第二版）	Data Structures and Problem Solving Using Java, 2E	Mark Allen Weiss	陈清亮	中、英
算法设计技巧与分析	Algorithms Design Techniques and Analysis	M. H. Alsuwaiyel	朱 洪	中、英
分布式算法导论（第二版）	Introduction to Distributed Algorithms, 2E	Gerard Tel		英
计算与算法导论	Introduction to Computing and Algorithms	Russell Shackelford	高小鹏	中
并行计算全书	The Sourcebook of Parallel Computing	Ian Foster	李晓梅	中
计算机系统结构：量化研究方法（第三版）	Computer Architecture: A Quantitive Approach, 3E	John L. Hennessy	郑纬民	中
现代处理器设计：超标量处理器基础	Modern Processor Design: Fundamentals of Superscalar Processors	John Paul Shen	戴 葵	中
Intel宏编语言程序设计（第四版）	Assembly Language for Intel-Based Computers, 4E	Kip R. Irvine	罗云彬	中
数字与微处理器基础：理论与应用（第四版）	Digital and Microprocessor Fundamentals: Theory and Application, 4E	William Kleitz	张大镭	中
计算机系统	Computer System: A Programmer's Perspective	Randal E. Bryant		英
Intel微处理器结构、编程与接口（第六版）	The Intel Microprocessors Architecture, Programming, and Interfacing, 6E	Barry B. Brey	金惠华	中
计算机系统设计与体系结构（第二版）	Computer Systems Design and Architecture, 2E	Vincent P. Heuring	邹恒明	中、英
操作系统：并发与分布式软件设计	Operating Systems: Concurrent and Distributed Software Design	Jean Bacon	陈向群	中、英
操作系统：内核与设计原理（第四版）	Operating Systems: Internals and Design Principles, 4E	William Stallings	郭 鹏	中
UNIX初级教程（第四版）	UNIX Unbounded: A Beginning Approach, 4E	Amir Afzal	张焕国	中、英
并行计算机互连网络技术：一种工程方法	Interconnection Networks: An Engineering Approach	Jose Duato	谢伦国	中
多媒体通信系统：技术、标准与网络	Multimedia Communication Systems: Techniques, Standards, and Networks	K. R. Rao	冯 刚	中
用TCP/IP进行网际互联（第一、二、三卷）	Internetworking with TCP/IP Vol 1,Vol 2,Vol 3	Douglas E. Comer	谢希仁	中
数据与计算机通信（第七版）	Data and Computer Communications, 7E	William Stallings	谢希仁	中
密码编码学与网络安全：原理与实践（第三版）	Cryptography and Network Security: Principles and Practice, 3E	Williams Stallings	张焕国	中
数据压缩原理与应用（第二版）	Data Compression: The Complete Reference, 2E	David Salomon	吴乐南	中
网络处理器与网络系统设计	Network System Design	Douglas E. Comer		英
网络安全：公众世界中的私人通信（第二版）	Network Security: Private Communication in a Public World, 2E	Charlie Kaufman	许剑卓	中
计算机安全学：安全的艺术与科学	Computer Security: Art and Science	Matt Bishop	陈克非	中
现代密码学理论与实践	Modern Cryptography: Theory & Practice	Wenbo Mao	王育民	中、英
网络分析、体系结构与设计（第二版）	Network Analysis, Architecture, and Design, 2E	James D. McCabe	李 腾	中
信息安全原理与应用（第三版）	Security in Computing, 3E	Charles P. Pfleeger	秦志光	中
C++大学教程	C++ How to Program	Harvey M. Deitel	邱仲潘	中
C++编程：从问题分析到程序设计	C++ Programming: From Problem Analysis to Program Design	D. S. Malik	钟书毅	中
C++编程导论（第二版）	An Introduction to Programming With C++, 2E	Diane Zak	马良荔	中
C++核心思想（第三版）	Computing Concepts with C++ Essentials, 3E	Cay Horstmann	晏海华	中
C++工程问题求解：面向对象方法	Engineering Problem Solving with C++: An Object-Oriented Approach	Delores M. Etter	周 靖	中
高级Java 2大学教程	Advanced Java 2 Platform How to Program	Harvey M. Deitel	钱 力	中、英
Java简明教程	Simply Java Programming	Harvey M. Deitel	张海恩	中
Java面向对象程序设计（第二版）	Java, Java, Java: Object-Oriented Problem Solving, 2E	Ralph Morelli	董永东	中
Java与UML面向对象设计与编程	Java the UML Way: Integrating Object-Oriented Design Programming	Else Lervik	王 箭	中
现代编译器的Java实现（第二版）	Modern Compiler Implementation in Java, 2E	Andrew W. Appel	程 虎	中
类型和程序设计语言	Types and Programming Languages	Benjamin C. Pierce	马世龙	中
程序设计语言基础	Foundations for Programming Languages	John C. Mitchell	许满武	中

国外计算机科学教材系列 ②

中文书名	英文书名	作者	译审者	版别
程序设计语言：原理与实践（第二版）	Programming Languages: Principles and Practice, 2E	Kenneth C. Louden	黄林鹏	中、英
C++面向对象编程（第二版）	Object-Oriented Programming Using C++, 2E	Ira Pohl		英
人机交互（第三版）	Human-Computer Interaction, 3E	Alan Dix Janet Finlay	蔡利栋	中、英
交互设计：超越人机交互	Interaction Design: Beyond Human-Computer Interaction	Jennifer Preece	麦中凡	中
用户界面设计（第三版）	Designing the User Interface: Strategies for Effective Human-Computer Interaction, 3E	Ben Shneiderma	张国印	中
数字图像处理	Digital Image Processing	K. R. Castleman	朱志刚	中
计算机图形学（第三版）	Computer Graphics, 3E	Donald Hearn	蔡士杰	中、英
计算机视觉：一种现代方法	Computer Vision: A Modern Approach	David A. Forsyth	林学訚	中
虚拟现实系统：接口、应用与设计	Understanding Virtual Reality: Interface, Application, and Design	William Sherman	吴玲达	中
非真实感图形学：造型、绘制与动画技术	Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation	Thomas Strothotte	叶修梓	中
虚拟现实技术（第二版）	Virtual Reality Technology, 2E	Grigore C. Burdea	吴玲达	中
人工智能：理论与实践	Artificial Intelligence: Theory and Practice	Thomas Dean	顾国昌	中、英
人工智能	Artificial Intelligence	Rob Callan	黄厚宽	中
多Agent系统引论	An Introduction to Multiagent Systems	Michael Wooldridge	石纯一	中
智能系统：结构、设计与控制	Intelligent Systems: Architecture, Design and Control	Alexander M. Meystel	冯祖仁	中、英
统计模式识别（第二版）	Statistical Pattern Recognition, 2E	Andrew Webb	边肇祺	中
模式识别（第二版）	Pattern Recognition, 2E	Sergios Theodoridis	李晶皎	中
统计自然语言学基础	Foundation of Statistical Natural Language Processing	Christopher D. Manning	苑春法	中
自然语言理解（第二版）	Natural Language Understanding, 2E	James F. Allen	刘 群	中
自然语言处理综论	Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition	Daniel Jurafsky	冯志伟	中
统计学习理论	Statistical Learning Theory	Vladimir Naumovich Vapnik	张学工	中
统计学习基础：数据挖掘、推理与预测	The Elements of Statistical Learning: Data Mining, Inference, and Prediction	Trevor Hastie	范明柴	中
支持向量机导论	An Introduction to Support Vector Machines and Other Kernel-based Learning Methods	Nello Cristianini	李国正	中
动态系统的反馈控制（第四版）	Feedback Control of Dynamic Systems, 4E	Gene F. Franklin	朱齐丹	中
机器人学导论：分析、系统与应用	Introduction to Robotics: Analysis, Systems, Applications	Saeed B. Niku	孙增圻	中
智能机器人导论	An Introduction to AI Robotics	Robin R. Murphy	孙增圻	中
机器人探索：工程实践技术手册	Robotic Explorations: A Hands-on Introduction to Engineering	Fred G. Martin	刘 荣	中
数据库设计、应用开发与管理（第二版）	Database Design, Application Development and Administration, 2E	Michael V. Mannino	唐常杰	中
现代数据库管理（第六版）	Modern Database Management, 6E	Jeffrey A. Hoffer		英
数据库：模型、语言与设计	Database: Models, Languages, Design	Jams L. Johnson	李天柱	中
数据库性能调优：原理与技术	Database Tuning: Principles, Experiments, and Troubleshooting Techniques	Dennis Shasha	孟小峰	中
面向对象的设计与模式	Object-Oriented Design and Patterns	Cay Horstmann	张泰恩	中
Petri网与系统工程：模拟、证明与应用指南	Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications	Claude Girault	袁崇义	中
软件质量工程：度量与模型（第二版）	Metrics and Models in Software Quality Engineering, 2E	Stephen H. Kan	何志均	中
B方法	The B-Book: Assigning Programs to Meanings	J-R Abrial	裘宗燕	中
C++数值算法（第二版）	Numerical Recipes in C++: The Art of Scientific Computing, 2E	William H. Press	胡健伟	中、英
网格计算（第二版）	The Grid 2: Blueprint for a New Computing Infrastructure	Ian Foster		中

● 更多图书信息，请登录华信教育资源网：www.huaxin.com.cn

● 详细的资料索取与教辅支持，请联系：010-88254555, 88254560, te_service@phei.com.cn





Visual Basic® .NET How to Program Second Edition

BOOK / CD-ROM

©2002, 1400 pp., paper (0-13-029363-6)

Learn Visual Basic .NET programming from the ground up! This book provides a comprehensive introduction to Visual Basic .NET—featuring extensive updates and increased functionality. *Visual Basic .NET How to Program, Second Edition* covers introductory programming techniques as well as more advanced topics, featuring enhanced treatment of developing Web-based applications. Other topics discussed include XML and wireless applications, databases, SQL and ADO .NET, Web forms, Web services and ASP .NET.



Internet & World Wide Web How to Program Third Edition

BOOK / CD-ROM

©2004, 1250 pp., paper (0-13-145091-3)

This book introduces students with little or no programming experience to the exciting world of Web-based applications. This text provides in-depth coverage of introductory programming principles, various markup languages (XHTML, Dynamic HTML and XML), several scripting languages (JavaScript, JScript .NET, ColdFusion, Flash ActionScript, Perl, PHP, VBScript and Python), Web servers (IIS and Apache) and relational databases (MySQL)—all the skills and tools needed to create dynamic Web-based applications. The text contains a comprehensive introduction to ASP .NET and the Microsoft .NET Framework. A case study illustrating how to build an online message board using ASP .NET and XML is also included. New in this edition are chapters on Macromedia ColdFusion, Macromedia Dreamweaver and a much enhanced treatment of Flash, including a case study on building a video game in Flash. After mastering the material in this book, students will be well prepared to build real-world, industrial-strength, Web-based applications.



Wireless Internet & Mobile Business How to Program

©2002, 1292 pp., paper (0-13-062226-5)

This book offers a thorough treatment of both the management and technical aspects of this growing area, including coverage of current practices and future trends. The first half explores the business issues surrounding wireless technology and mobile business. The book then turns to programming for the wireless Internet, exploring topics such as WAP (including 2.0), WML, WMLScript, XML, XHTML™, wireless Java programming (J2ME™) and more. Other topics covered include career resources, wireless marketing, accessibility, Palm™, PocketPC, Windows CE, i-mode, Bluetooth, MIDP, MIDlets, ASP, Microsoft .NET Mobile Framework, BREW™, multimedia, Flash™ and VBScript.



Python How to Program

BOOK / CD-ROM

©2002, 1376 pp., paper (0-13-092361-3)

This exciting textbook provides a comprehensive introduction to Python—a powerful object-oriented programming language with clear syntax and the ability to bring together various technologies quickly and easily. This book covers introductory programming techniques and more advanced topics such as graphical user interfaces, databases, wireless Internet programming, networking, security, process management, multithreading, XHTML, CSS, PSP and multimedia. Readers will learn principles that are applicable to both systems development and Web programming.



e-Business & e-Commerce for Managers

©2001, 794 pp., cloth (0-13-032364-0)

This comprehensive overview of building and managing e-businesses explores topics such as the decision to bring a business online, choosing a business model, accepting payments, marketing strategies and security, as well as many other important issues (such as career resources). The book features Web resources and online demonstrations that supplement the text and direct readers to additional materials. The book also includes an appendix that develops a complete Web-based shopping-cart application using HTML, JavaScript, VBScript, Active Server Pages, ADO, SQL, HTTP, XML and XSL. Plus, company-specific sections provide "real-world" examples of the concepts presented in the book.

Sign up now for the FREE *Deitel® Buzz Online* newsletter at:
www.deitel.com/newsletter/subscribe.html



XML How to Program

BOOK / CD-ROM

©2001, 934 pp., paper (0-13-028417-3)

This book is a comprehensive guide to programming in XML. It teaches how to use XML to create customized tags and includes chapters that address markup languages for science and technology, multimedia, commerce and many other fields. Concise introductions to Java, JavaServer Pages, VBScript, Active Server Pages and Perl/CGI provide readers with the essentials of these programming languages and server-side development technologies to enable them to work effectively with XML. The book also covers topics such as XSL, DOM™, SAX, a real-world e-commerce case study and a complete chapter on Web accessibility that addresses VoiceXML. Other topics covered include XHTML, CSS, DTD, schema, parsers, XPath, XLink, namespaces, XBase, Xinclude, XPointer, XSLT, XSL Formatting Objects, JavaServer Pages, XForms, topic maps, X3D, MathML, OpenMath, CML, BML, CDF, RDF, SVG, Cocoon, WML, XBRL and BizTalk™ and SOAP™ Web resources.



Perl How to Program

BOOK / CD-ROM

©2001, 1057 pp., paper (0-13-028418-1)

This comprehensive guide to Perl programming emphasizes the use of the Common Gateway Interface (CGI) with Perl to create powerful, dynamic multi-tier Web-based client/server applications. The book begins with a clear and careful introduction to programming concepts at a level suitable for beginners, and proceeds through advanced topics such as references and complex data structures. Key Perl topics such as regular expressions and string manipulation are covered in detail. The authors address important and topical issues such as object-oriented programming, the Perl database interface (DBI), graphics and security. Also included is a treatment of XML, a bonus chapter introducing the Python programming language, supplemental material on career resources and a complete chapter on Web accessibility.



e-Business & e-Commerce How to Program

BOOK / CD-ROM

©2001, 1264 pp., paper (0-13-028419-X)

This book explores programming technologies for developing Web-based e-business and e-commerce solutions, and covers e-business and e-commerce models and business issues. Readers learn a full range of options, from "build-your-own" to turnkey solutions. The book examines scores of the top e-businesses (examples include Amazon, eBay, Priceline, Travelocity, etc.), explaining the technical details of building successful e-business and e-commerce sites and their underlying business premises. Learn how to implement the dominant e-commerce models—shopping carts, auctions, name-your-own-price, comparison shopping and bots/intelligent agents—by using markup languages (HTML, Dynamic HTML and XML), scripting languages (JavaScript, VBScript and Perl), server-side technologies (Active Server Pages and Perl/CGI) and database (SQL and ADO), security and online payment technologies.



Visual Basic® 6 How to Program

BOOK / CD-ROM

©1999, 1016 pp., paper (0-13-456955-5)

ORDER INFORMATION

For ordering information, visit us on the Web at www.prenhall.com.

INTERNATIONAL ORDERING INFORMATION

CANADA:
Pearson Education Canada
26 Prince Andrew Place
PO Box 580
Don Mills, Ontario M3C 2T8 Canada
Tel.: 416-925-2249; Fax: 416-925-0068
e-mail: phcinfo.pubcanada@pearsoned.com

EUROPE, MIDDLE EAST, AND AFRICA:
Pearson Education
Edinburgh Gate
Harlow, Essex CM20 2JE UK
Tel: 01279 623928; Fax: 01279 414130
e-mail: enq.orders@pearsoned-ema.com

BENELUX REGION:
Pearson Education
Concertgebouwplein 25
1071 LM Amsterdam
The Netherlands
Tel: 31 20 5755 800; Fax: 31 20 664 5334
e-mail: amsterdam@pearsoned-ema.com

ASIA:
Pearson Education Asia Pte. Ltd.
23/25 First Lok Yang Road
Jurong, 629733 Singapore
Tel: 65 476 4688; Fax: 65 378 0370

JAPAN:
Pearson Education Japan
Ogikubo TM Bldg. 6F, 5-26-13 Ogikubo
Suginami-ku, Tokyo 167-0051 Japan
Tel: 81 3 3365 9001; Fax: 81 3 3365 9009

INDIA:
Pearson Education
Indian Branch
482 FIE, Patparganj
Delhi – 110092 India
Tel: 91 11 2059850 & 2059851
Fax: 91 11 2059852

AUSTRALIA:
Pearson Education Australia
Unit 4, Level 2, 14 Aquatic Drive
Frenchs Forest, NSW 2086, Australia
Tel: 61 2 9454 2200; Fax: 61 2 9453 0089
e-mail: marketing@pearsoned.com.au

NEW ZEALAND/FIJI:
Pearson Education
46 Hillside Road
Auckland 10, New Zealand
Tel: 649 444 4968; Fax: 649 444 4957
E-mail: sales@pearsoned.co.nz

SOUTH AFRICA:
Maskew Miller Longman
Central Park Block H
16th Street Midrand 1685
South Africa
Tel: 27 21 686 6356; Fax: 27 21 686 4590

LATIN AMERICA:
Pearson Education Latin America
Attn: Tina Sheldon
1 Lake Street
Upper Saddle River, NJ 07458

The DEITEL® Suite of Products...

HOW TO PROGRAM BOOKS



C++ How to Program Fifth Edition

BOOK / CD-ROM

©2005, 1500 pp., paper (0-13-185757-6)

The complete authoritative DEITEL® *LIVE-CODE* introduction to programming with C++! The Fifth Edition takes a new easy-to-follow, carefully developed early classes and early objects approach to programming in C++. The text includes comprehensive coverage of the fundamentals of object-oriented programming in C++. It includes a a new optional automated teller machine (ATM) case study that teaches the fundamentals of software engineering and object-oriented design with the UML 2.0 in Chapters 1-7, 9 and 13. Additional integrated case studies appear throughout the text, including the `Time` class (Chapter 9), the `Employee` class (Chapters 12 and 13) and the `GradeBook` class (Chapters 3-7). The book also includes a new interior design including updated colors, new fonts, new design elements and more.



Small C++ How to Program Fifth Edition

BOOK / CD-ROM

©2005, 900 pp., paper (0-13-185758-4)

Based on chapters 1-13 (except the optional OOD/UML case study) and appendices of *C++ How to Program, Fifth Edition*, *Small C++* features a new early classes and early objects approach and comprehensive coverage of the fundamentals of object-oriented programming in C++. Key topics include applications, variables, memory concepts, data types, control statements, functions, arrays, pointers and strings, inheritance and polymorphism.

Coming in Spring 2005 for both *C++ How to Program, 5/e* and *Small C++ How to Program, 5/e*: FREE C++ Web-based *Cyber Classroom* with the purchase of a new textbook. The *Cyber Classroom* includes a complete e-book, audio walkthroughs of the code examples, a FREE Lab Manual and selected student solutions. See pages 7–9 of this advertorial for more information.



Java™ How to Program Sixth Edition

BOOK / CD-ROM

©2005, 1500 pp., paper (0-13-148398-6)

The complete authoritative DEITEL® *LIVE-CODE* introduction to programming with the new Java™ 2 Platform Standard Edition 5.0! New early classes and early objects approach. *Java How to Program, Sixth Edition* is up-to-date with J2SE™ 5.0 and includes comprehensive coverage of the fundamentals of object-oriented programming in Java; a new interior design including new colors, new fonts, new design elements and more; and a new optional automated teller machine (ATM) case study that teaches the fundamentals of software engineering and object oriented design with the UML 2.0 in Chapters 1-8 and 10. Additional integrated case studies appear throughout the text, including GUI and graphics (Chapters 3-12), the `Time` class (Chapter 8), the `Employee` class (Chapters 9 and 10) and the `GradeBook Employee` (Chapters 3-8). New J2SE 5.0 topics covered included input/output, enhanced `for` loop, autoboxing, generics, new collections APIs and more.



Small Java™ How to Program, Sixth Edition

BOOK / CD-ROM

©2005, 700 pp., paper (0-13-148660-8)

Based on chapters 1-10 of *Java™ How to Program, Sixth Edition*, *Small Java* is up-to-date with J2SE™ 5.0, features a new early classes and early objects approach and comprehensive coverage of the fundamentals of object-oriented programming in Java. Key topics include applications, variables, data types, control statements, methods, arrays, object-based programming, inheritance and polymorphism.

Coming in Spring 2005 for both *Java How to Program, 6/e* and *Small Java How to Program, 6/e*: Free Java Web-based *Cyber Classroom* with the purchase of a new textbook. The *Cyber Classroom* includes a complete e-book, audio walkthroughs of the code examples, a FREE Lab Manual and selected student solutions. See pages 7–8 of this advertorial for more information.

Sign up now for the FREE *DEITEL® BUZZ ONLINE* newsletter at:
www.deitel.com/newsletter/subscribe.html



C How to Program Fourth Edition

BOOK / CD-ROM

©2004, 1255 pp., paper
(0-13-142644-3)

C How to Program, Fourth Edition—the world's best-selling C text—is designed for introductory through intermediate courses as well as programming languages survey courses. This comprehensive text is aimed at readers with little or no programming experience through intermediate audiences. Highly practical in approach, it introduces fundamental notions of structured programming and software engineering and gets up to speed quickly.

A Student Solutions Manual is also available is for use with this text. Use ISBN 0-13-145245-2 to order.



Getting Started with Microsoft® Visual C++™ 6 with an Introduction to MFC

BOOK / CD-ROM

©2000, 163 pp., paper
(0-13-016147-0)



Visual C++ .NET® How To Program

BOOK / CD-ROM

©2004, 1400 pp., paper
(0-13-437377-4)

Written by the authors of the world's best-selling introductory/intermediate C and C++ textbooks, this comprehensive book thoroughly examines Visual C++® .NET. *Visual C++® .NET How to Program* begins with a strong foundation in the introductory and intermediate programming principles students will need in industry, including fundamental topics such as arrays, functions and control statements. Readers learn the concepts of object-oriented programming. The text then explores such essential topics as networking, databases, XML and multimedia. Graphical user interfaces are also extensively covered, giving students the tools to build compelling and fully interactive programs using the "drag-and-drop" techniques provided by Visual Studio .NET 2003.



Advanced Java™ 2 Platform How to Program

BOOK / CD-ROM

©2002, 1811 pp., paper
(0-13-089560-1)

Expanding on the world's best-selling Java textbook—*Java™ How to Program*—*Advanced Java™ 2 Platform How To Program* presents advanced Java topics for developing sophisticated, user-friendly GUIs; significant, scalable enterprise applications; wireless applications and distributed systems. Primarily based on Java 2 Enterprise Edition (J2EE), this textbook integrates technologies such as XML, JavaBeans, security, JDBC™, JavaServer Pages (JSP™), servlets, Remote Method Invocation (RMI), Enterprise JavaBeans™ (EJB), design patterns, Swing, J2ME™, Java 2D and 3D, XML, design patterns, CORBA, Jini™, JavaSpaces™, Jiro™, Java Management Extensions (JMX) and Peer-to-Peer networking with an introduction to JXTA.



C# How to Program

BOOK / CD-ROM

©2002, 1568 pp., paper
(0-13-062221-4)

C# How to Program provides a comprehensive introduction to Microsoft's C# object-oriented language. C# enables students to create powerful Web applications and components—ranging from XML-based Web services on Microsoft's .NET platform to middle-tier business objects and system-level applications. *C# How to Program* begins with a strong foundation in the introductory- and intermediate-programming principles students will need in industry. It then explores such essential topics as object-oriented programming and exception handling. Graphical user interfaces are extensively covered, giving readers the tools to build compelling and fully interactive programs. Internet technologies such as XML, ADO .NET and Web services are covered as well as topics including regular expressions, multithreading, networking, databases, files and data structures.



MULTIMEDIA CYBER CLASSROOMS

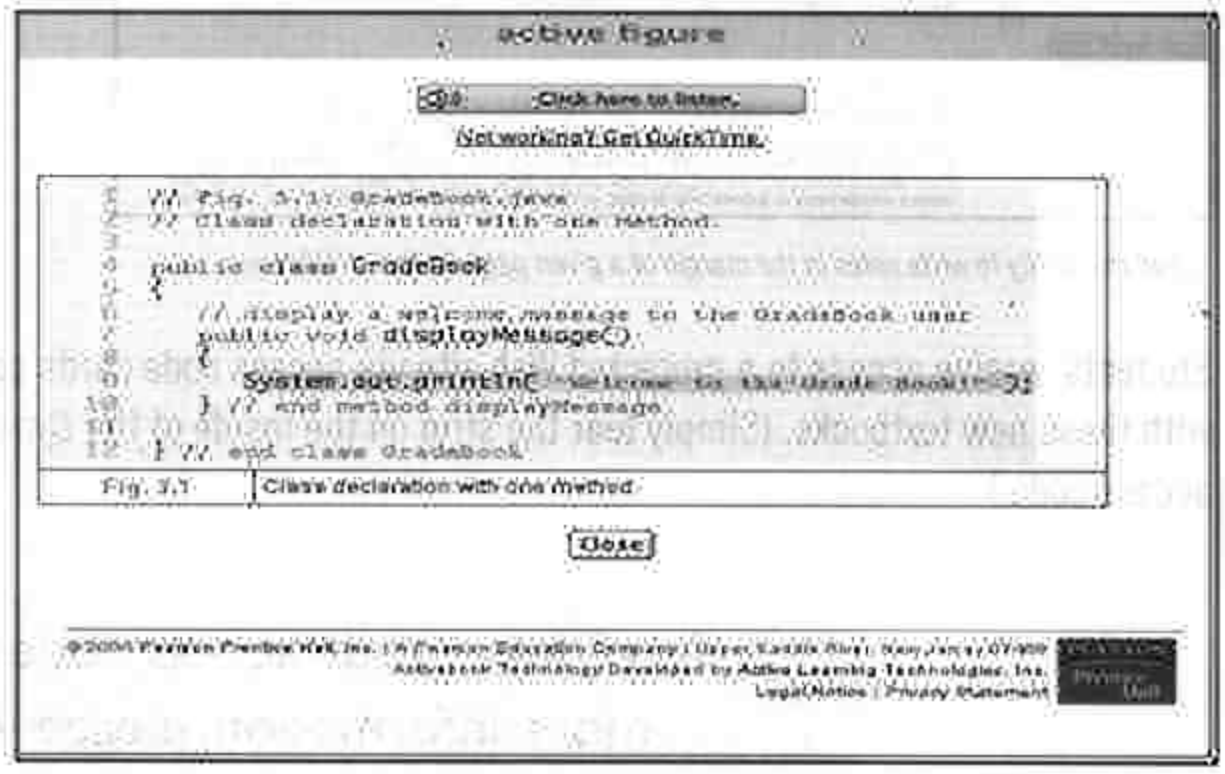
Premium content available FREE with *Java™* and *Small Java™ How to Program, Sixth Edition* and *C++* and *Small C++ How to Program, Fifth Edition!*

Java and *Small Java How to Program, 6/e* and *C++* and *Small C++ How to Program, 5/e* will soon be available with a **FREE** Web-based *Multimedia Cyber Classroom* for students who purchase new copies of these books! The *Cyber Classroom* is an interactive, multimedia, tutorial version of DEITEL textbooks. *Cyber Classrooms* are a great value, giving students additional hands-on experience and study aids.



COMING SOON
for Java and Small Java How to Program, 6/e and C++ and Small C++ How to Program, 5/e (with purchase of book)

DEITEL® Multimedia Cyber Classrooms *feature the complete text of their corresponding* How to Program *titles.*



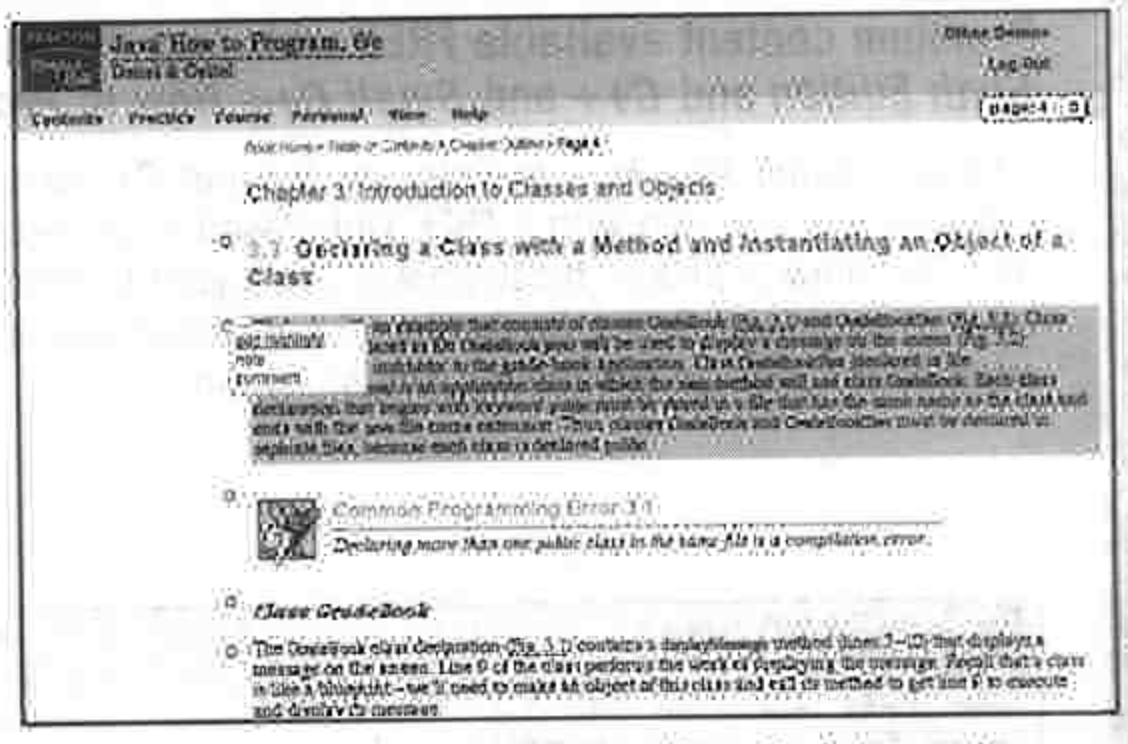
Unique audio "walkthroughs" of code examples reinforce key concepts.

Sign up now for the FREE *DEITEL® Buzz Online* newsletter at:
www.deitel.com/newsletter/subscribe.html

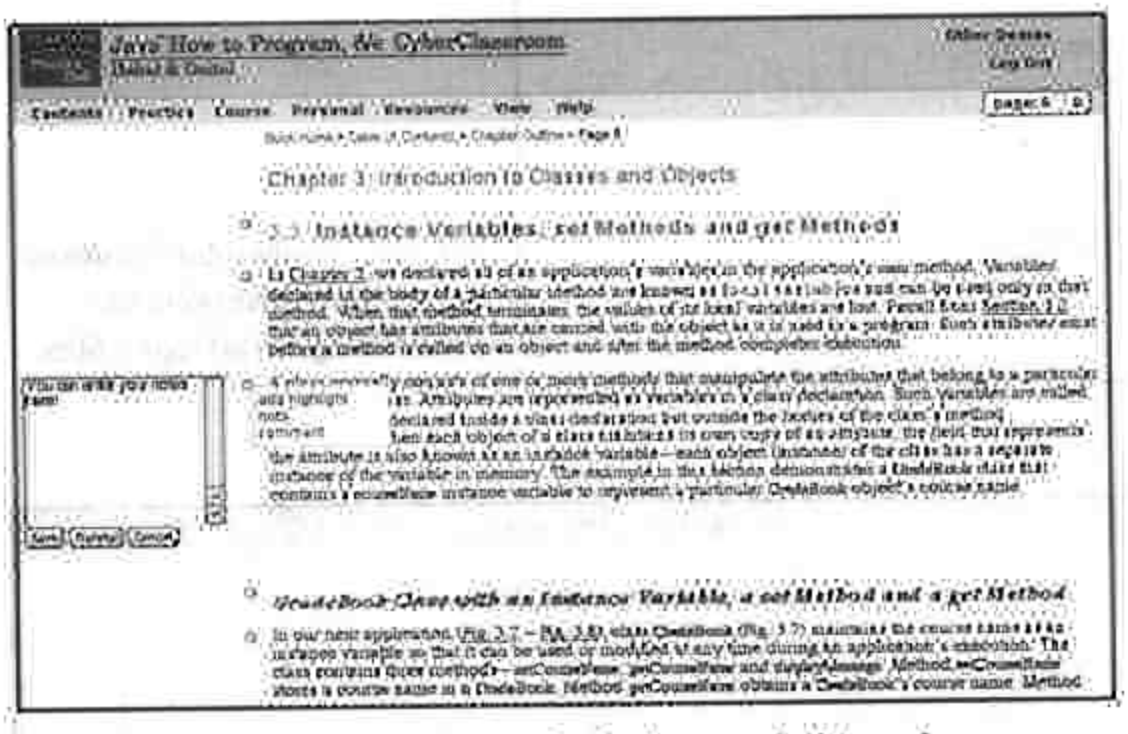
MULTIMEDIA CYBER CLASSROOMS

DEITEL® *Multimedia Cyber Classrooms* include:

- The full text, illustrations and program listings of its corresponding *How to Program* book.
- Hours of detailed, expert audio descriptions of hundreds of lines of code that help to reinforce important concepts.
- An abundance of self-assessment material, including practice exams, hundreds of programming exercises and self-review questions and answers.



DEITEL® Multimedia Cyber Classrooms offer a host of interactive features, such as highlighting of key sections of the text...



...and the ability to write notes in the margin of a given page for future reference.

- Intuitive browser-based interface designed to be easy and accessible.
- A Lab Manual featuring lab exercises as well as pre- and post-lab activities.
- Student Solutions to approximately one-half of the exercises in the textbook.

Students receive access to a protected Web site via access code cards packaged, for FREE, automatically with these new textbooks. (Simply tear the strip on the inside of the *Cyber Classroom* package to reveal access code.)

To redeem your access code or for more information, please visit:
www.prenhall.com/deitel/cyberclassroom

The SIMPLY SERIES!

The Deitels' *Simply Series* takes an engaging new approach to teaching programming languages from the ground up. The pedagogy of this series combines the DEITEL® signature *LIVE-CODE Approach* with an *APPLICATION-DRIVEN Tutorial Approach* to teach programming with outstanding pedagogical features that help students learn. We have merged the notion of a lab manual with that of a conventional textbook, creating a book in which readers build and execute complete applications from start to finish, while learning the fundamental concepts of programming!



Simply C++ An APPLICATION-DRIVEN Tutorial Approach

©2005, 800 pp., paper (0-13-142660-5)

Simply C++ An APPLICATION-DRIVEN Tutorial Approach guides readers through building real-world applications that incorporate C++ programming fundamentals. Learn methods, functions, data types, control statements, procedures, arrays, object-oriented programming, strings and characters, pointers, references, templates, operator overloading and more in this comprehensive introduction to C++.



Simply Java™ Programming An APPLICATION-DRIVEN Tutorial Approach

©2004, 950 pp., paper (0-13-142648-6)

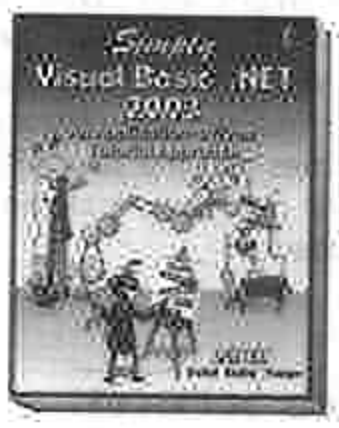
Simply Java™ Programming An APPLICATION-DRIVEN Tutorial Approach guides readers through building real-world applications that incorporate Java programming fundamentals. Learn GUI design, components, methods, event-handling, types, control statements, arrays, object-oriented programming, exception-handling, strings and characters, sequential files and more in this comprehensive introduction to Java. We also include higher-end topics such as database programming, multimedia, graphics and Web applications development.



Simply C# An APPLICATION-DRIVEN Tutorial Approach

©2004, 860 pp., paper (0-13-142641-9)

Simply C# An APPLICATION-DRIVEN Tutorial Approach guides readers through building real-world applications that incorporate C# programming fundamentals. Learn GUI design, controls, methods, functions, data types, control statements, procedures, arrays, object-oriented programming, strings and characters, sequential files and more in this comprehensive introduction to C#. We also include higher-end topics such as database programming, multimedia and graphics and Web applications development.



Simply Visual Basic® .NET An APPLICATION-DRIVEN Tutorial Approach

Visual Studio .NET 2002 Version: *©2003, 830 pp., paper (0-13-140553-5)*

Visual Studio .NET 2003 Version: *©2004, 960 pp., paper (0-13-142640-0)*

Simply Visual Basic® .NET An APPLICATION-DRIVEN Tutorial Approach guides readers through building real-world applications that incorporate Visual Basic .NET programming fundamentals. Learn GUI design, controls, methods, functions, data types, control statements, procedures, arrays, object-oriented programming, strings and characters, sequential files and more in this comprehensive introduction to Visual Basic .NET. We also include higher-end topics such as database programming, multimedia and graphics and Web applications development. If you're using Visual Studio® .NET 2002, choose *Simply Visual Basic .NET*; or, if you're using Visual Studio .NET 2003, you can use *Simply Visual Basic .NET 2003*, which includes updated screen captures and line numbers consistent with Visual Studio .NET 2003.

Sign up now for the FREE *DEITEL® BUZZ ONLINE* newsletter at:
www.deitel.com/newsletter/subscribe.html

The DEITEL® DEVELOPER SERIES!

Deitel & Associates is recognized worldwide for its best-selling *How to Program Series* of books for college and university students and its signature *LIVE-CODE Approach* to teaching programming languages. Now, for the first time, Deitel & Associates brings its proven teaching methods to a series of books specifically designed for professionals.

THREE TYPES OF BOOKS FOR THREE DISTINCT AUDIENCES

A Technical Introduction

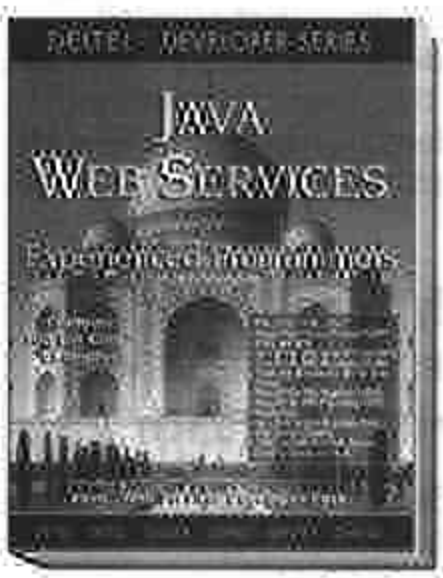
A Technical Introduction books provide programmers, technical managers, project managers and other technical professionals with introductions to broad new technology areas.

A Programmer's Introduction

A Programmer's Introduction books offer focused treatments of programming fundamentals for practicing programmers. These books are also appropriate for novices.

For Experienced Programmers

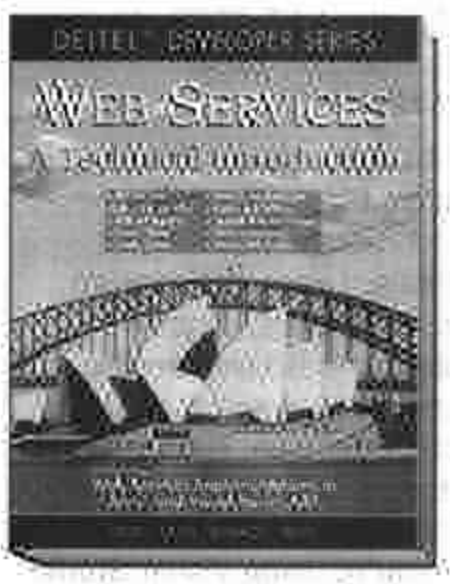
For Experienced Programmers books are for experienced programmers who want a detailed treatment of a programming language or technology. These books contain condensed introductions to programming language fundamentals and provide extensive intermediate level coverage of high-end topics.



Java™ Web Services for Experienced Programmers

©2003, 700 pp., paper (0-13-046134-2)

Java™ Web Services for Experienced Programmers covers industry standards including XML, SOAP, WSDL and UDDI. Learn how to build and integrate Web services using the Java API for XML RPC, the Java API for XML Messaging, Apache Axis and the Java Web Services Developer Pack. Develop and deploy Web services on several major Web services platforms. Register and discover Web services through public registries and the Java API for XML Registries. Build Web services clients for several platforms, including J2ME. Significant Web services case studies also are included.



Web Services: A Technical Introduction

©2003, 400 pp., paper (0-13-046135-0)

Web Services: A Technical Introduction familiarizes programmers, technical managers and project managers with key Web services concepts, including what Web services are and why they are revolutionary. The book covers the business case for Web services—the underlying technologies, ways in which Web services can provide competitive advantages and opportunities for Web services-related lines of business. Readers learn the latest Web-services standards, including XML, SOAP, WSDL and UDDI; learn about Web services implementations in .NET and Java; benefit from an extensive comparison of Web services products and vendors; and read about Web services security options. Although this is not a programming book, the appendices show .NET and Java code examples to demonstrate the structure of Web services applications and documents. In addition, the book includes numerous case studies describing ways in which organizations are implementing Web services to increase efficiency, simplify business processes, create new revenue streams and interact better with partners and customers.

DIVE INTO™ SERIES CORPORATE TRAINING DELIVERED WORLDWIDE

Deitel & Associates, Inc. provides intensive, lecture-and-laboratory courses to organizations worldwide. The programming courses use our signature *Live-Code Approach*, presenting complete working programs.

Deitel & Associates, Inc. has trained over one million students and professionals worldwide through Dive Into Series® corporate training courses, public seminars, university teaching, *How to Program Series* textbooks, *Deitel® Developer Series* books, *Simply Series* textbooks, *Cyber Classroom Series* multimedia packages, *Complete Training Course Series* textbook and multimedia packages, broadcast-satellite courses and Web-based training.

Educational Consulting

Deitel & Associates, Inc. offers complete educational consulting services for corporate training programs and professional schools including:

- Curriculum design and development
- Preparation of Instructor Guides
- Customized courses and course materials
- Design and implementation of professional training certificate programs
- Instructor certification
- Train-the-trainers programs
- Delivery of software-related corporate training programs

Visit our Web site for more information on our Dive Into™ Series corporate training curriculum and to purchase our training products.

`www.deitel.com/training`

Would you like to review upcoming publications?

If you are a professor or senior industry professional interested in being a reviewer of our forthcoming publications, please contact us by email at **`deitel@deitel.com`**. Insert "Content Reviewer" in the subject heading.

Are you interested in a career in computer education, publishing and training?

We offer a limited number of full-time positions available for college graduates in computer science, information systems, information technology and management information systems. Please check our Web site for the latest job postings or contact us by email at **`deitel@deitel.com`**. Insert "Full-time Job" in the subject heading.

Are you a Boston-area college student looking for an internship?

We have a limited number of competitive summer positions and 20-hr./week school-year opportunities for computer science, IT/IS and MIS majors. Students work at our worldwide headquarters west of Boston. We also offer full-time internships for students taking a semester off from school. This is an excellent opportunity for students looking to gain industry experience and earn money to pay for school. Please contact us by email at **`deitel@deitel.com`**. Insert "Internship" in the subject heading.

Would you like to explore contract training opportunities with us?

Deitel & Associates, Inc. is looking for contract instructors to teach software-related topics at our clients' sites in the United States and worldwide. Applicants should be experienced professional trainers or college professors. For more information, please visit **`www.deitel.com`** and send your resume to Abbey Deitel at **`abbey.deitel@deitel.com`**.

Are you a training company in need of quality course materials?

Corporate training companies worldwide use our *How to Program Series* textbooks, *Complete Training Course Series* book and multimedia packages, *Simply Series* textbooks and our *Deitel® Developer Series* books in their classes. We have extensive ancillary instructor materials for many of our products. For more details, please visit **`www.deitel.com`** or contact us by email at **`deitel@deitel.com`**.

PROGRAMMING LANGUAGE TEXTBOOK AUTHORS

DEITEL & ASSOCIATES, INC.

Check out our Dive Into Series™ Corporate On-site Seminars...

Java™

- Java for Nonprogrammers
- Java for VB/COBOL Programmers
- Java for C/C++ Programmers

Advanced Java™

- Java™ Web Services
- J2ME™
- J2EE
- Enterprise JavaBeans (EJB™) and Design Patterns
- Advanced Swing GUI
- RMI
- JDBC
- CORBA
- JavaBeans™

Internet & World Wide Web Programming

- Client-Side Internet & World Wide Web Programming
- Server-Side Internet & World Wide Web Programming
- Perl/CGI, Python, PHP

C/C++

- C and C++ Programming: Part 1 (for Nonprogrammers)
- C and C++ Programming: Part 2 (for Non-C Programmers)
- C++ and Object-Oriented Programming
- Advanced C++ and Object-Oriented Programming

XML

- XML Programming for programmers with Java, Web or other programming experience

.NET Programming

- C# Programming
- Visual Basic .NET Programming
- Visual C++ .NET Programming

Other Topics

- Object-Oriented Analysis and Design with the UML
- SQL Server
- e-Business and e-Commerce

For Detailed Course Descriptions, Visit Our Web Site: www.deitel.com

Through our worldwide network of trainers, we would be happy to attempt to arrange corporate on-site courses for you in virtually any software-related field.

For Additional Information about Our Dive Into Series™ Corporate On-Site Courses, contact:

Abbey Deitel, President
Email: abbey.deitel@deitel.com
Phone: (978) 461-5880/Fax: (978) 461-5884

Our training clients include:

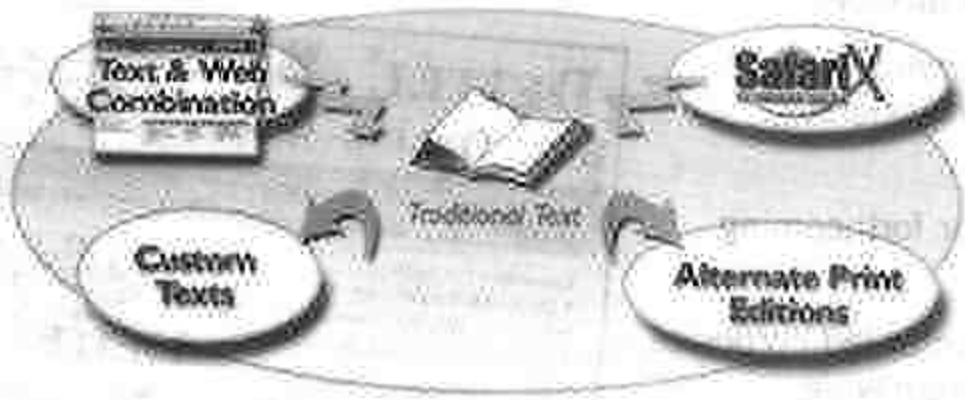
3Com
Argonne National Laboratories
Art Technology
Avid Technology
Bank of America
BEA Systems
BlueCross BlueShield
Boeing
Bristol-Myers Squibb
Cambridge Technology Partners
Cap Gemini
Concord Communications
Dell
Dunn & Bradstreet
Eastman Kodak
EMC²
Federal Reserve Bank of Chicago
Fidelity
GE
General Dynamics Electric Boat Corporation
Gillette
GTE
Hitachi
IBM
Invensys
JPL Laboratories
Lockheed Martin
Lucent
MapInfo Corporation
MCI
Motorola
NASA's Kennedy Space Center
NASDAQ
National Oceanographic and Atmospheric Administration
Nortel Networks
Omnipoint
One Wave
Open Market
Oracle
Pacific Capital Bank Corporation
PalmSource
Pragmatics
Primavera
Progress Software
Rogue Wave Software
Schlumberger
Sun Microsystems
Symmetrix
Telefónica
Thompson Technology
Tivoli Systems
TJX Companies
Toys "R" Us
U.S. Army at Ft. LeaVenworth
Visa International
Washington Post Newsweek Interactive
White Sands Missile Range
and more!

www.deitel.com www.prenhall.com/deitel

PearsonChoices

For Instructors and Students using DEITEL® Publications!

Today's students have increasing demands on their time and money, and they need to be resourceful about how, when and where they study. Pearson Education has responded to that need by creating PearsonChoices, which allows faculty and students to choose from a variety of formats and prices.



Visit www.pearsonchoices.com for more information!

We are pleased to announce PearsonChoices this fall for our brand new DEITEL publications:

- Small Java How to Program, Sixth Edition and Small C++ How to Program, Fifth Edition—our alternative print editions to *Java How to Program, Sixth Edition* and *C++ How to Program, Fifth Edition* at a competitive price!

Small Java How to Program and *Small C++ How to Program* bring the solid and proven pedagogy of our fully updated *Java How to Program 6/E* and *C++ How to Program 5/E* to new, smaller texts that are purely focused on CS1 courses and priced lower than our full and comprehensive *How to Program* texts and other competing texts in the CS1 market. See the first page of this advertorial for more on these compelling new Java and C++ titles!

- SafariX WebBooks—We are pleased to offer students five NEW DEITEL SafariX WebBooks available for Fall 2005 at 50% off the print version's price!
 - *Java How to Program, Sixth Edition*
 - *Small Java How to Program, Sixth Edition*
 - *C++ How to Program, Fifth Edition*
 - *Small C++ How to Program, Fifth Edition*
 - *Simply C++: An Application-Driven Tutorial Approach*



SafariX TEXTBOOKS ONLINE

SafariX Textbooks Online is an exciting new service for college students looking to save money on required or recommended textbooks for academic courses!

This secure WebBooks platform creates a new option in the higher education market, an additional choice for students alongside conventional textbooks and online learning services. By eliminating the costs relating to printing, manufacturing and retail distribution for the physical textbook, Pearson provides students with a WebBook at 50% of the cost of its conventional print equivalent. Beginning spring 2005, students can choose to purchase a print edition textbook or subscribe to the same textbook content through SafariX Textbooks Online.

SafariX WebBooks are digital versions of print textbooks enhanced with features such as high-speed search, note taking and bookmarking. SafariX WebBooks are viewed through a Web browser connected to the Internet. SafariX Textbooks Online requires no special plug-ins and no applications download to your computer. Students just log in, purchase access and begin studying!

With SafariX Textbooks Online students will be able to search the text, make notes online, print out reading assignments that incorporate your lecture notes and bookmark important passages they want to review later. They can navigate easily to a page number, reading assignment or chapter. The Table of Contents of each WebBook appears in the left hand column alongside the text.

Visit www.safarix.com for more information!

Sign up now for the FREE *Deitel® Buzz Online* newsletter at:
www.deitel.com/newsletter/subscribe.html

DEITEL® BUZZ ONLINE NEWSLETTER

The *DEITEL® BUZZ ONLINE* e-mail newsletter is now sent to over 38,000 opt-in subscribers. This free publication is designed to keep you updated on our publishing program, instructor-led corporate training courses, the latest industry topics and trends and more.

Each issue of our newsletter includes:

- Highlights and announcements on our products available for professionals, students and instructors.
- Sample chapters from our forthcoming publications.
- Information on our instructor-led corporate training courses taught worldwide.
- Detailed ordering information for instructors, professors, students and professionals, along with additional book resources and downloads.





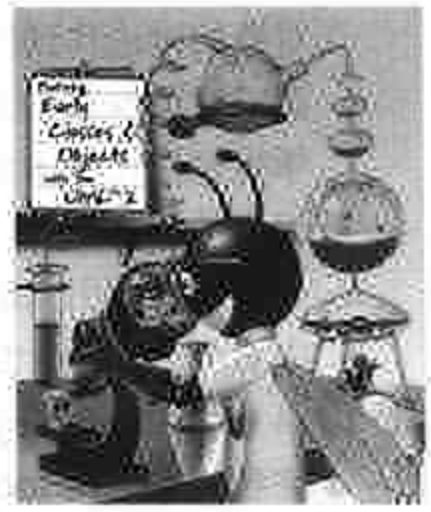
- Available in both full-color HTML or plain-text format.
- Approximately 6–8 issues per year.
- Your e-mail address will not be distributed or sold.

Turn the page to find out more about Deitel & Associates!



To sign up for the *DEITEL® BUZZ ONLINE* newsletter, visit www.deitel.com/newsletter/subscribe.html.

前　言



"诚实地对待每个细节，这正是你能力之所在。"

——修女 Teresa

欢迎大家进入C++编程世界并学习本书。C++是什么？简单地说，C++是一门全球性的程序设计语言，主要用于并发工业级、高性能的计算机应用程序。为此，我们编写了大量适于高校计算机科学教学的编程教科书和专业参考书，不断地在 Deitel & Associates 出版公司出版。本书正是我们的成果之一，编写过程令人愉快。它基于我们已大幅度修订了内容的《C++大学教程》（第五版），是为学制为一学期的基础课程（CS1，Computer Science 1）提供的一本简洁的教材。本书集中介绍C++核心概念和特性，覆盖了《C++大学教程》（第五版）前13章的内容。

我们相信这本教材及其辅助材料包罗了教师和学生进行C++教学所需要的一切，使他们充分体验集信息性、趣味性、挑战性和娱乐性于一体的学习历程。在本前言中，我们首先概述本书的新特性。而在前言的"本书导读"部分，我们帮助教师、学生和专业人士了解本书中C++和面向对象编程知识的覆盖程度。另外，在前言中，我们还将说明本书中所用的各种约定，如"代码清洗"（code washing）和代码用粗体显示等。最后，我们提供网上免费编译器的信息，并详述一整套协助教师更好地开展教学的教学资源信息，包括教师资源光盘、PowerPoint课件幻灯片的授课备注、课程管理系统和 SafariX（Pearson 教育的网络书籍出版物）等。

本书的新特性

为了编写本书，我们仔细研究了《C++大学教程》的以往版本。这一新版本具有许多引人注目的特性：

- **大幅度的内容修订。**所有的章节都经过了有意义的校正和提升。我们不断改进，使得语言表达更为清晰与准确。同时，我们也调整了对C++术语的表达方式，使其与定义该语言的ANSI/ISO C++标准文档相一致。
- **更小的章节。**较大的章节分割成了更易于管理的若干较小的章节（例如，第四版的第1章被分割成第1章和第2章，第四版的第2章是现在这一版的第4章和第5章）。
- **早期类和对象的涉及。**在教学程序上，我们做了一些变动，就是在教学的早期就开始涉及类和对象的内容。在第1章，就向学生介绍对象技术的基本概念和术语。在以前的版本中，学生在第6章才开始开发自定义的、可重用的类及对象。但是，在新的版本中，在全新的第3章中就开始这项工作。第4章到第7章也是从"尽早接触类和对象"这一观点出发，我们做了认真的改写。这个新版本是面向对象的，无论从书的开篇，还是贯穿整本书。把关于类和

序时，可以确认程序是否如期望的那样运行。将输出结果与产生这些结果的程序语句关联起来，对于概念的学习和巩固，将起到意想不到的作用。我们的示例程序展示了C++的各种特性。这些代码都注明了行数，这样使代码阅读更加容易——当学生阅读本书中更大型的程序时，会更加体会到这样编排的优势所在。

402幅插图/图表

本书包含了大量的图表、表格、线绘图、程序及程序输出。我们采用UML活动图表示控制语句中的控制流，用UML类图来表示类的数据成员、构造函数和成员函数。

403条编程提示

本书还包含了很多编程提示，目的在于帮助学生将学习重点放在程序开发的关键部分。为了突出这些提示，我们将它们分了类，具体如下：良好的编程习惯、常见的编程错误、性能提示、可移植性提示、软件工程知识和错误预防技巧。这些提示和实践经验是我们60年的编程和教学经验的总结。我们的一个数学专业的学生说，这种方法就像在数学中着重标明公理、定理、引理和推论一样，同样为编写一个好的软件提供了基础。



良好的编程习惯

良好的编程习惯是一些关于编写清晰程序的提示。这些技术有助于学生编写出更易阅读、更易修改和更易于维护的程序。



常见的编程错误

编程或程序设计语言的初学者总是犯一些常见的错误。关注我们这里总结的常见错误能减少犯类似错误的可能性，也可以缩短老师办公室门前问问题的学生队伍的长度。



性能提示

根据我们的经验，对于第一门程序设计语言课程而言，教授学生编写清晰的、易于理解的程序是迄今为止最重要的目标。但是学生往往喜欢编写速度奇快、内存占用最少、击键次数最少或使用令人刮目相看的其他技巧的程序。实际上，他们关心的是性能问题。他们想知道自己的程序经过哪些调整之后便可以改善性能。因此，我们提供了一些强调提高程序性能的方法，使程序运行得更快，或者它们所占用的内存最少。



可移植性提示

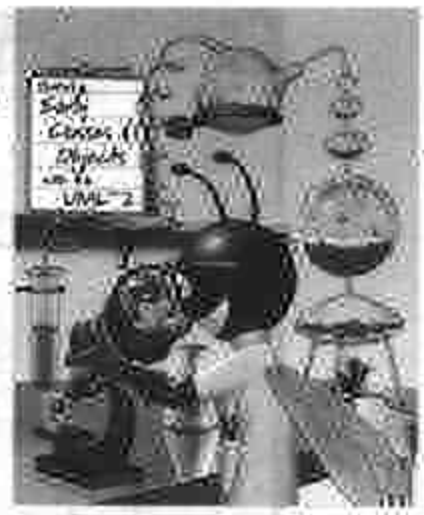
软件开发是一个复杂而又昂贵的过程。软件开发组织通常需要开发不同版本的软件来适应不同类型的计算机和操作系统的需要。所以，今天我们特别重视程序的可移植性。换言之，一个软件应该只需少量修改（或者根本不用修改），便可以在其他不同的计算机系统上运行。一些程序员认为，只要他们采用标准C++来实现应用程序，那么应用程序就是可移植的了。但是，事实并非如此简单。要想达到可移植性，需要非常仔细而谨慎的设计。通常存在很多陷阱。我们书中所包含的可移植性提示就是为了帮助学生编写可移植的代码，同时也使大家更好地领悟C++是如何实现其高度可移植性的。



软件工程知识

面向对象编程方法要求我们完全重新考虑软件系统构建的方式。C++是一种可以实现良好的软件工程的有效语言。软件工程知识这一部分的提示突出了关于软件体系结构和设计的问题。这些问题往往影响到整个软件系统的构架，特别对于一些大的系统而言。学生在此学到的大多数知识，对于他们高级课程的学习，或者当他们走上工作岗位开始接触大型、复杂的现实系统时非常有用。

学前准备



在读者开始使用本书之前，请仔细阅读这里的用法说明，以保证书中范例的正确复制。

字体与命名的约定

我们通过字体将屏幕成分（如菜单名和菜单项）同C++代码或者命令区分开来。具体的约定是：采用sans-serif Helvetica粗体突出屏幕成分，例如File菜单，而用sans-serif Lucida字体强调C++代码和命令，比如`cout << "Hello";`。

本书附带光盘中的资源

本书附带的光盘包括如下内容：

- 数以百计的C++活代码范例。
- 免费C++编辑器以及集成开发环境（IDE）的链接。
- 丰富的Web资源，包含全面的参考书目、教程、常见问题解答（FQA）和新闻组。

无论有什么问题，都欢迎大家随时通过电子邮件与我们联系（电子邮件的地址为deitel@deitel.com），我们会迅速地给予答复。

文件的复制与组织

本书的所有范例都包含在随书赠送的光盘中。请遵照“从光盘复制本书范例”一节的方法将这张光盘examples目录下的内容复制到你的硬盘中。在此建议大家学习的时候使用硬盘而非光盘的原因有两点：一是这张光盘是只读性的，因此读者无法将自己的应用程序保存其中；二是文件从硬盘的读取速度要比从光盘的读取速度快。另外，书中的范例也可以从以下地址下载：

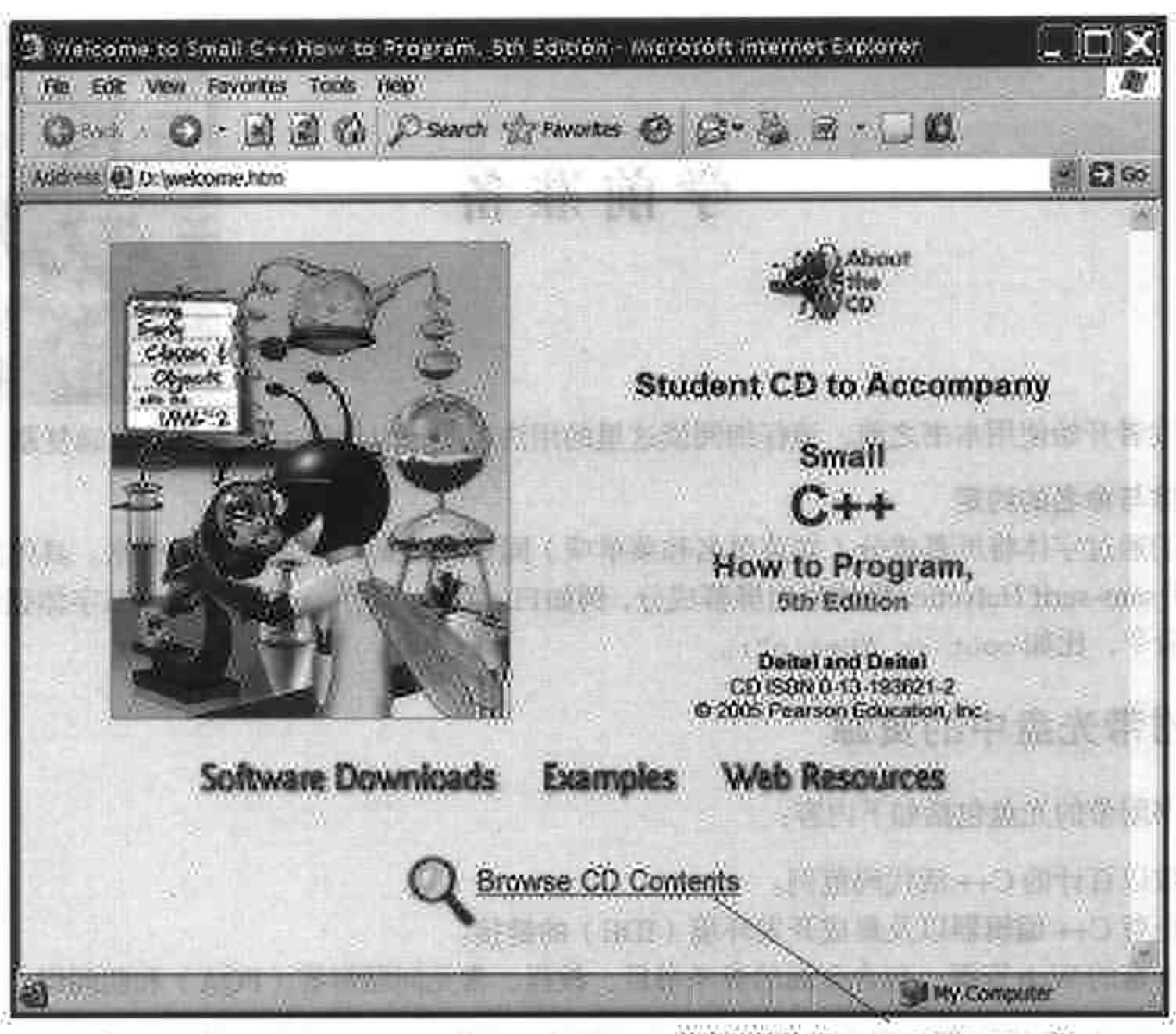
www.deitel.com/books/scpphtp5/index.html

www.prenhall.com/deitel

为了介绍学前准备，我们假定读者所用计算机运行的是Microsoft Windows系统。在接下来的小节中，所显示的屏幕截图很可能和实际计算机上看到的稍有不同，取决于读者使用的是Windows 2000还是Windows XP。如果读者的计算机运行的是其他不同的操作系统，且在向计算机复制范例文件时存在问题，则请找教师解答。

从光盘复制本书范例

1. 插入光盘。将本书附带的光盘插入计算机的光盘驱动器后，不久会出现如图1所示的窗口。如果显示了该页面，则转到本节的第3步继续进行；如果未显示该页面，则继续第2步。



单击此处的 "Browse CD Contents"

图1　本书光盘的欢迎页面

2. 利用 My Computer（我的电脑）打开光盘目录。如果如图 1 所示的页面没有出现，那么请双击桌面上的 My Computer（我的电脑）图标。接着，在 My Computer（我的电脑）窗口中双击你的光盘驱动器（如图 2 所示），载入这张光盘，结果如图 1 所示。

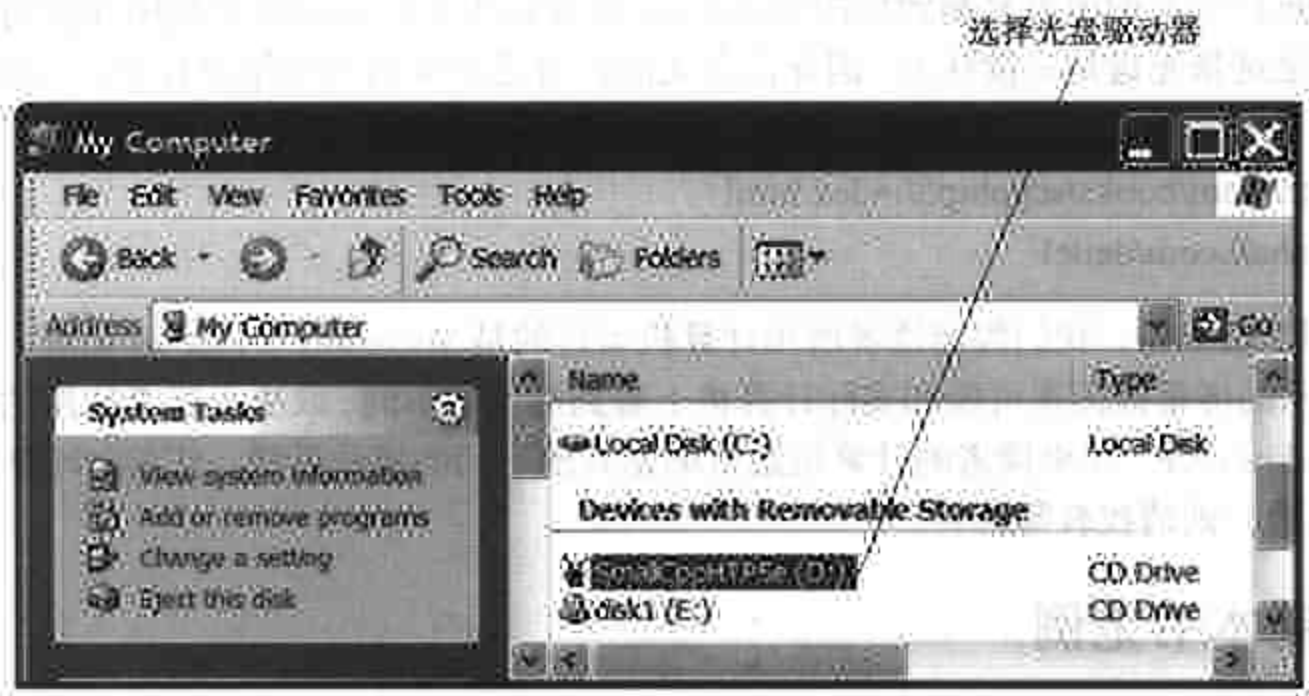


图2　定位到光盘驱动器

3. 打开光盘目录。如果如图1所示的页面确实出现了，那么请单击Browse CD Contents（浏览光盘内容）链接（如图1所示），访问光盘内容。
4. 复制examples目录。如图3所示，右击要重新复制的examples目录，然后选择Copy（复制）。接着到My Computer，双击盘符C:。选择Edit（编辑）菜单的Paste（粘贴）选项，复制光盘上的examples目录及其内容到C:盘。请注意：我们在此将范例保存到C:盘，并在整本教材中使用该路径。大家可以根据自己的计算机设置、学校实验室的情况或者个人的喜好，选择用于保存你的复制文件的不同驱动器。如果你打算在计算机实验室学习，请进一步征求教师以确认保存范例的具体地点。

由光盘复制到你的计算机上的本书范例文件是只读文件，所以还需消除只读属性，这样才可以修改和运行它们。

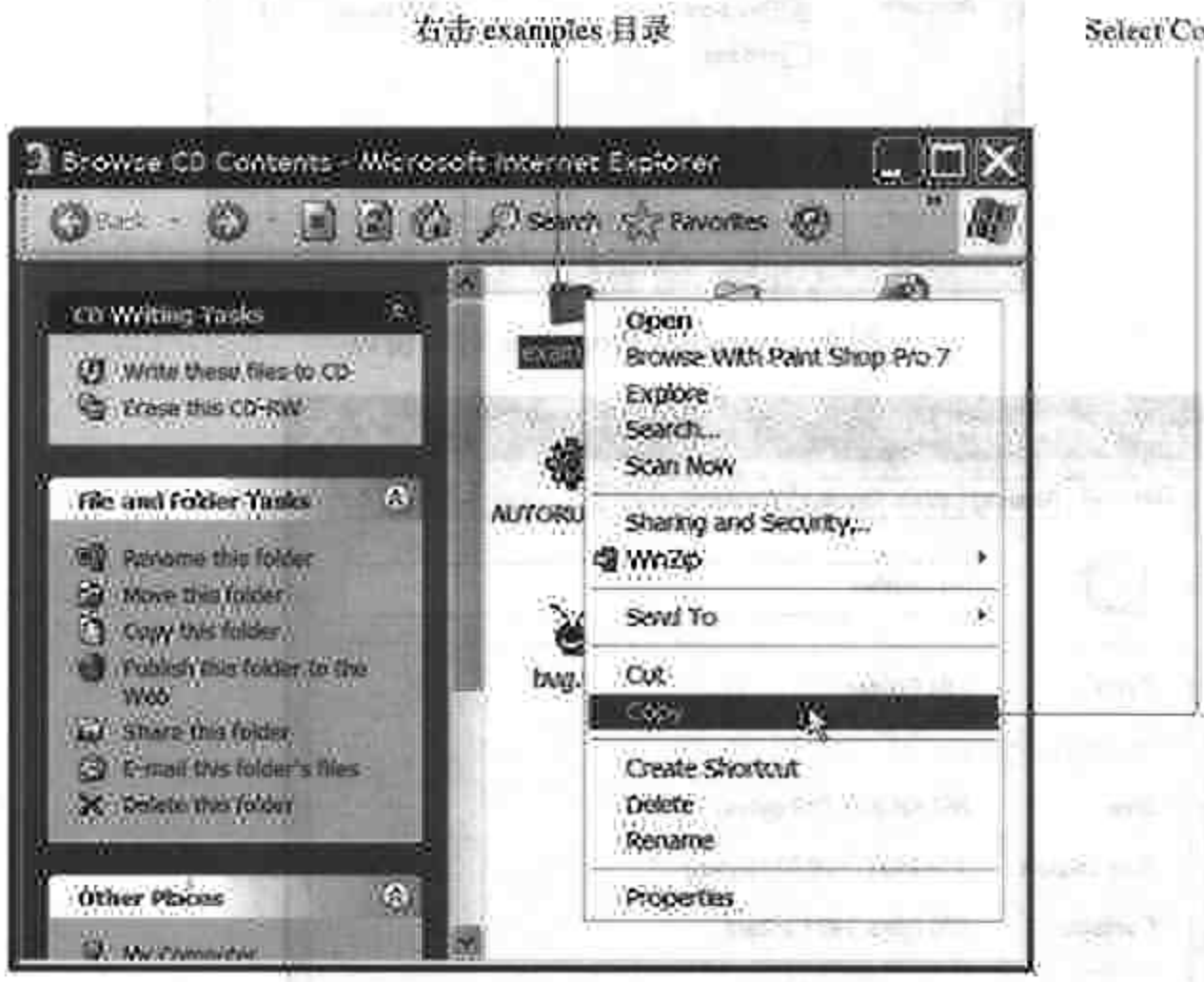


图3　复制examples目录

更改文件的只读属性

1. 打开Properties（属性）对话窗口。右击examples目录，从菜单上选择Properties（属性）后，出现如图4所示的examples Properties对话窗口。
2. 改变只读属性。在这个对话窗口的Attributes（属性）部分，单击与Read-only（只读）连接的复选框，取消如图5所示的复选标记。单击Apply（应用）使该改动产生作用。
3. 改变所有文件的属性。单击Apply将显示如图6所示的Confirm Attribute Changes（确认属性更改）窗口。在此窗口中，单击与Apply changes to this folder, subfolders and files（将更改应用于这个文件夹、子文件夹和文件）连接的单选按钮，然后单击OK（确定）按钮，消除examples目录中所有文件和目录的只读属性。



图 4　examples Properties 对话窗口

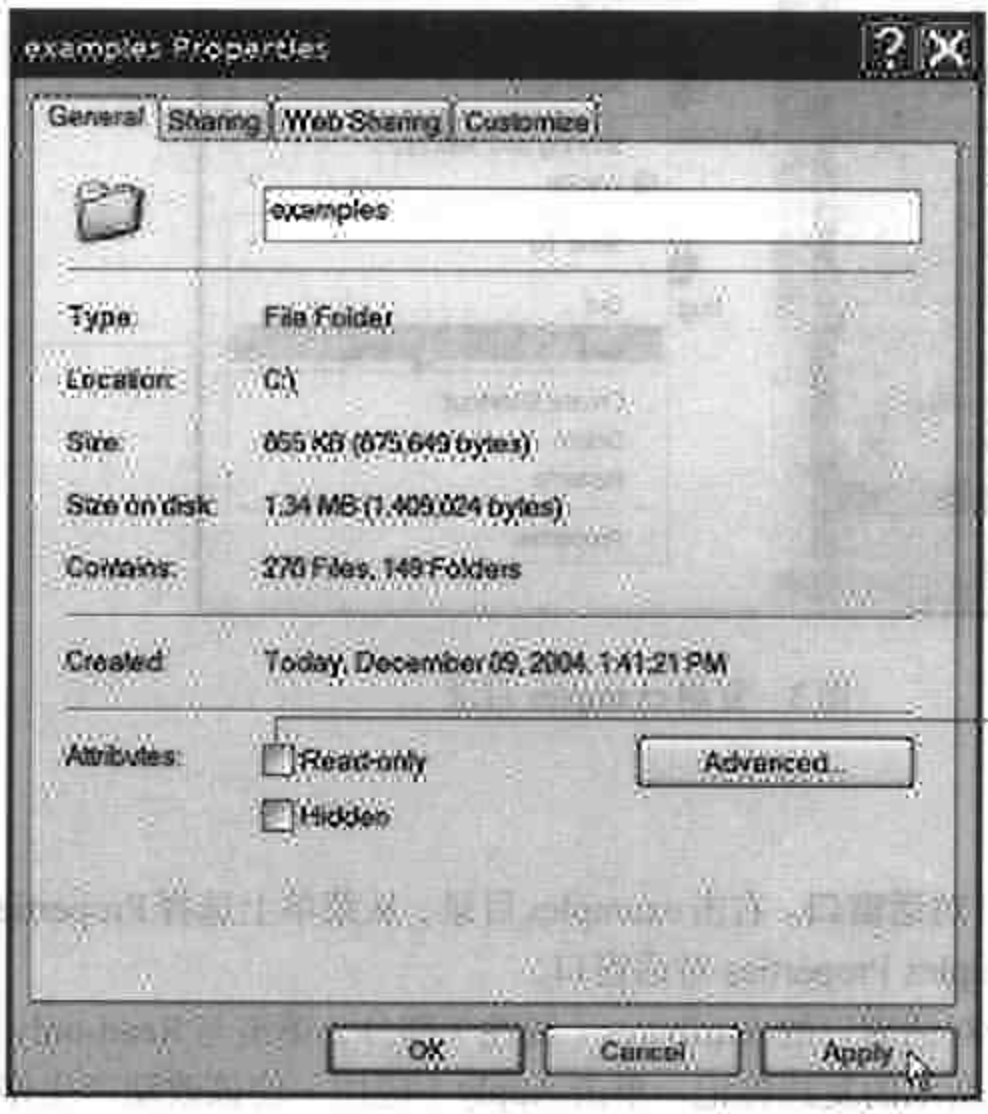


图 5　取消选中 Read-only 复选框

现在，读者就可开始自己的 C++ 学习之旅，我们衷心祝愿你旅途愉快！你可以随时通过 deitel@deitel.com 与我们联系，我们会迅速地给予答复。

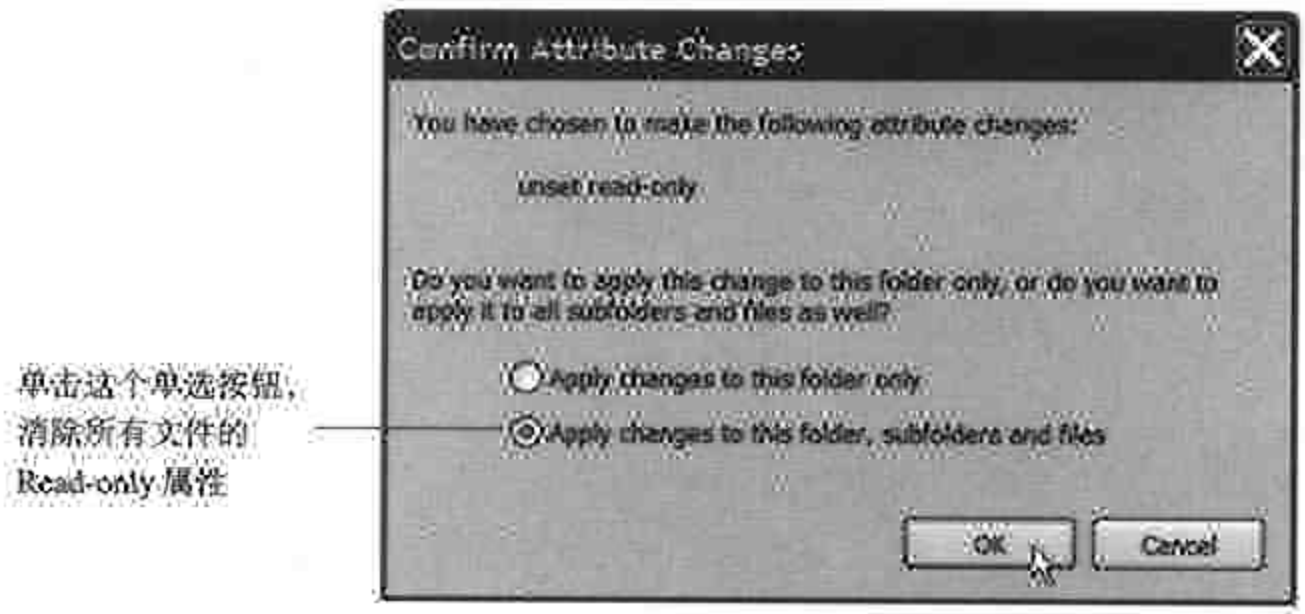


图.6　消除 examples 目录中所有文件的只读属性



第1章　计算机、互联网和万维网简介

语言的要领是清晰明白。

——Galen

我们的生活被细节所困扰……力求简洁，再简洁。

——Henry David Thoreau

他在思想的归纳和深入浅出的表达方面具有与众不同的才能。

——Thomas B. Macaulay

人类仍然是最超凡的计算机。

——John F. Kennedy

学习目标

在本章中将学习：

- 硬件和软件的基本概念
- 对象技术的基本概念，诸如类、对象、属性、行为、封装和继承等
- 不同类型的程序设计语言
- 哪些程序设计语言应用最广
- 一个典型的 C++ 程序开发环境
- 行业标准的面向对象系统建模语言 UML 的历史
- 互联网和万维网的历史
- 在两种流行的 C++ 环境（Linux 上运行的 GNU C++ 和 Windows XP 上运行的 Microsoft 公司的 Visual C++ .NET）中试运行 C++ 应用程序

提纲

1.1　简介
1.2　什么是计算机
1.3　计算机组成
1.4　早期的操作系统
1.5　个人计算、分布式计算和客户 / 服务器计算
1.6　互联网和万维网
1.7　机器语言、汇编语言和高级语言
1.8　C 和 C++ 的历史
1.9　C++ 标准库
1.10　Java 的历史
1.11　FORTRAN、COBOL、Pascal 和 Ada
1.12　Basic、Visual Basic、Visual C++、C# 和.NET
1.13　主要的软件趋势: 对象技术
1.14　典型的 C++ 开发环境
1.15　关于 C++ 和本书的几点说明
1.16　试运行一个 C++ 应用程序
1.17　对象技术和 UML 的介绍
1.18　本章小结

1.9　C++ 标准库

C++ 程序由一系列称为类（class）和函数（function）的构件组成。你可以编写构成一个 C++ 程序所需要的每个构件。不过，大多数C++程序员都会利用C++标准库（C++ Standard Library）所提供的丰富而现成的类和函数。所以，要想了解 C++ “世界”，实际上需要学习两方面的知识。一是学习 C++ 语言本身；二是学习使用 C++ 标准库所提供的各种类和函数。我们在这本书里会讨论很多类和函数。对于程序员，若想要深入理解包括在C++中的ANSI C库函数，思考如何实现它们，以及如何利用它们编写可移植代码的话，那么P J. Plauger的“The Standard C Library”（Upper Saddle River, NJ: Prentice Hall PTR, 1992）是必读之书。标准类库一般是由编译器厂商提供的，很多特殊用途的类库则由独立的软件商提供。



软件工程知识 1.1

使用“块构建方法”来创建程序。避免重复工作，尽量利用现有的构件。这就是软件重用（software reuse），也是面向对象编程的核心思想。



软件工程知识 1.2

在进行C++编程时，你通常会用到以下构件：源自 C++ 标准库中的类和函数、你和你的同事自己创建的类和函数、一些知名的第三方软件商提供的类和函数。

我们在整本书中提供了许多软件工程知识（Software Engineering Observation），目的是进一步解释一些影响和改善软件系统体系结构和质量的概念。我们也突出了各种其他类型的提示，包括良好的编程习惯（Good Programming Practice）、常见的编程错误（Common Programming Error）、性能提示（Performance Tip）、可移植性提示（Portability Tip）和错误预防技巧（Error-Prevention Tip）等。良好的编程习惯帮助大家编写出的程序更清楚易懂、更易测试和调试（debug）——或者说更易于去除程序的错误；常见的编程错误指出了应提防和避免的问题；性能提示提供了能够编写出运行得更快的程序和内存使用得更少的程序的各种技巧；可移植性提示给出了使你的程序不需要或只要一点点修改，就可以在各种计算机上运行的技术——这些提示还包括了C++如何做到高度可移植性的一般知识；错误预防技巧帮助你消除程序中的错误，更重要的是，有助于你从一开始就编写出无错程序。当然，很多只是指导方针，你还要发掘出自己的更好的编程风格。

自己创建函数和类的优点是可以确切地知道它们是怎样工作的，而且可以检查 C++ 代码。缺点是保证新函数和类的设计、开发和维护的正确性和操作有效性是费时和代价高的。



性能提示 1.1

因为 C++ 标准库函数和类是为了保证高效运行而经过精心设计才编写出来的，所以采用 C++ 标准库函数和类，而不是自己编写相应的版本，可以有效地提高程序的性能。这个小技巧也可以缩短程序开发的时间。



可移植性提示 1.2

因为每个 C++ 实现方案中都包含了 C++ 标准库函数和类，所以采用 C++ 标准库函数和类，而不是自己编写相应的版本，可以提高程序的可移植性。

1.10　Java 的历史

微处理器正在对智能消费电子设备领域产生深远的影响。为此，Sun Microsystems公司在1991年投资启动了一个代号为Green的内部合作研究项目。该项目最终开发出了一种基于C++语言的语言。有趣的是，该语言的创建者 James Gosling 最初以 Oak（橡树）来命名此语言，因为他在 Sun 公

书讨论了一些最常见的预处理指令。在第3阶段，编译器将C++程序翻译成机器语言代码（即目标代码）。

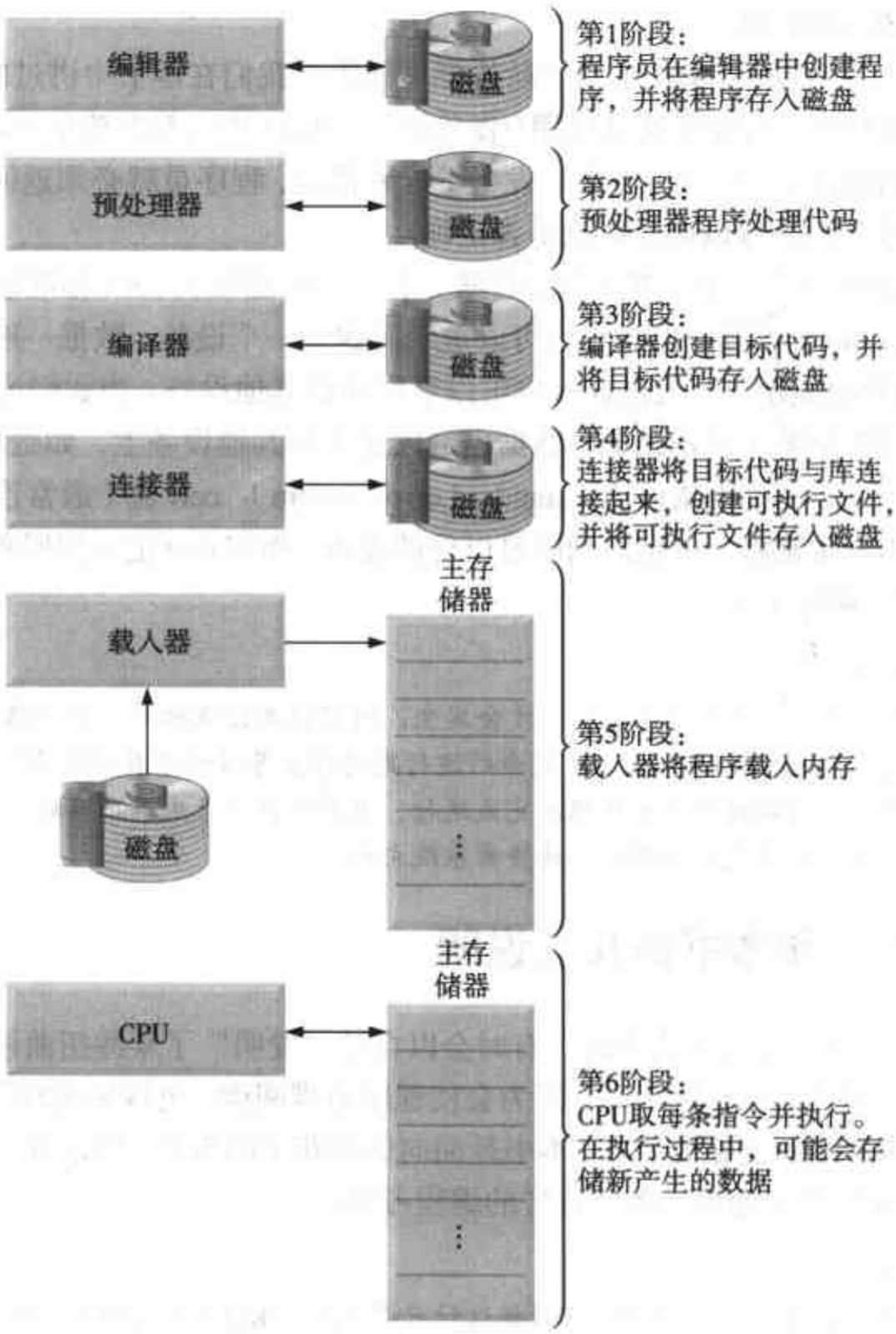


图 1.1　典型的 C++ 环境

第 4 阶段：连接

第4阶段是连接（linking）阶段。C++程序一般都包含了在别处定义的函数和数据的引用。例如，在标准库或者私有库（由参与某特定项目的程序员小组所创建）中定义的函数和数据。由于缺少这些引用的定义，C++编译器产生的目标代码通常包含了“洞”。连接器（linker）则将目标代码和这些缺少的函数代码连接起来，形成可执行程序（executable image）（其中不再有缺失的部分）。如果程序编译和连接都没有问题的话，就会生成可执行程序。

第 5 阶段：载入

第5阶段称为载入（loading）。在一个程序可以执行之前，必须先将它放入内存中。这由载入器（loader）来完成。载入器首先从磁盘中取得可执行程序，然后将其传输到内存。支持该程序的其他共享库组件在需要时，载入器也会载入。

“命令提示符”（Command Prompt）。要转到已完成的 GuessNumber（猜数）应用程序所在的目录，请输入 cd C:\examples\ch01\GuessNumber\ Windows，然后按回车键（如图 1.2 所示）。cd 命令是用来改变目录的。

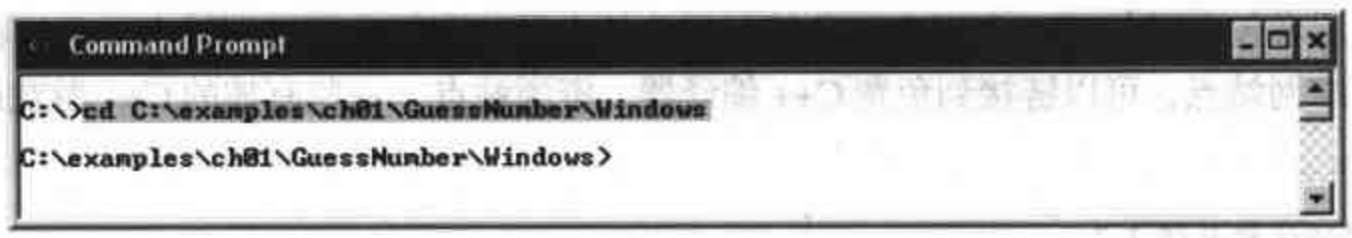


图 1.2 打开命令提示符窗口并改变目录

3. **运行 GuessNumber 应用程序。**既然你已经在含有 GuessNumber 应用程序的目录下，那么请输入命令 GuessNumber（如图 1.3 所示）并按回车键。注意：GuessNumber.exe 是这个应用程序的实际名字；尽管 Windows 假定.exe 是默认的扩展名。

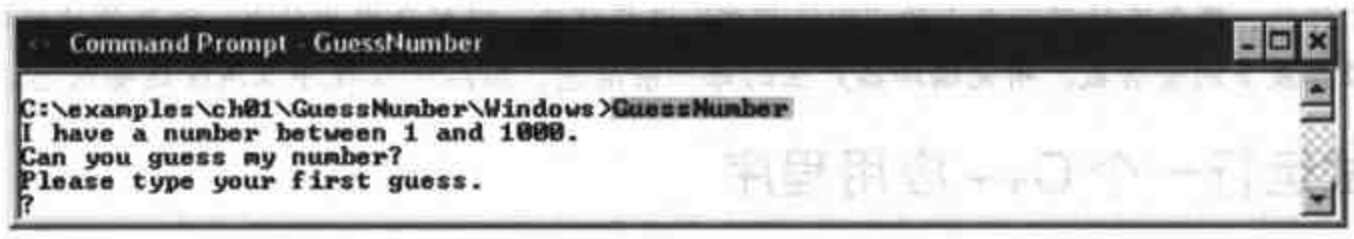


图 1.3 运行 GuessNumber 应用程序

4. **输入你第一次猜测的数。**应用程序显示 “Please type your first guess.”（请输入你猜测的第一个数），然后在下一行显示作为提示符的一个问号（?）（如图 1.3 所示）。在此提示符下，输入 500（如图 1.4 所示）。

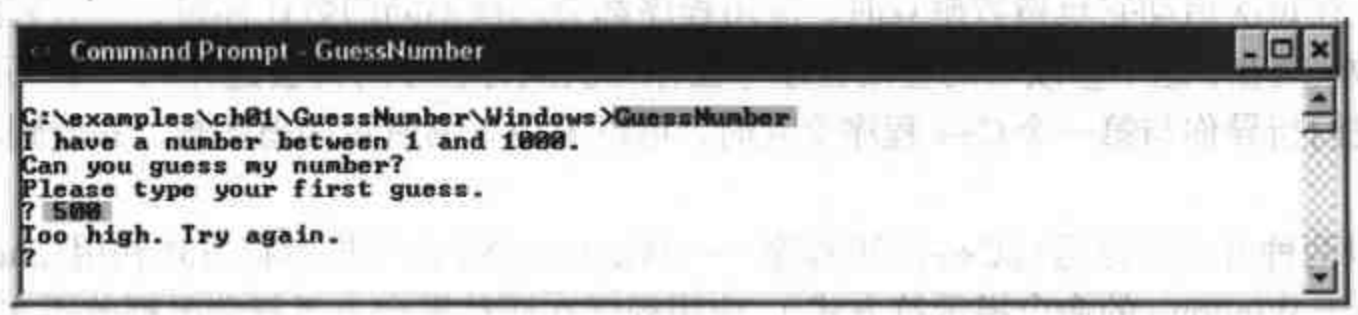


图 1.4 输入你第一次猜测的数

5. **输入下一次猜测的数。**应用程序显示 “Too high. Try again.”（太大了，再试一次），说明你输入的数比程序所选的正确数要大。因此，你接下来应该输入一个小一点的数。在提示下输入 250（如图 1.5 所示）后，应用程序又显示 “Too high. Try again.”（太大了，再试一次），因为你输入的数仍然比正确答案大。

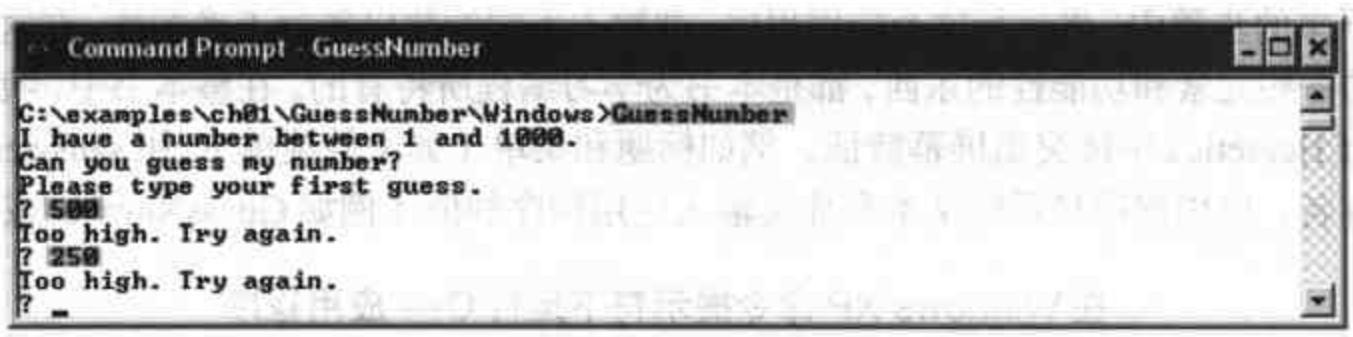


图 1.5 输入下一次猜测的数并收到反馈

6. **输入其他猜测的数。**输入数继续玩这个游戏，直到你猜对为止。一旦猜对了，应用程序会显示 “Excellent! You guessed the number!”（“非常棒！你猜对了！”）（如图 1.6 所示）。

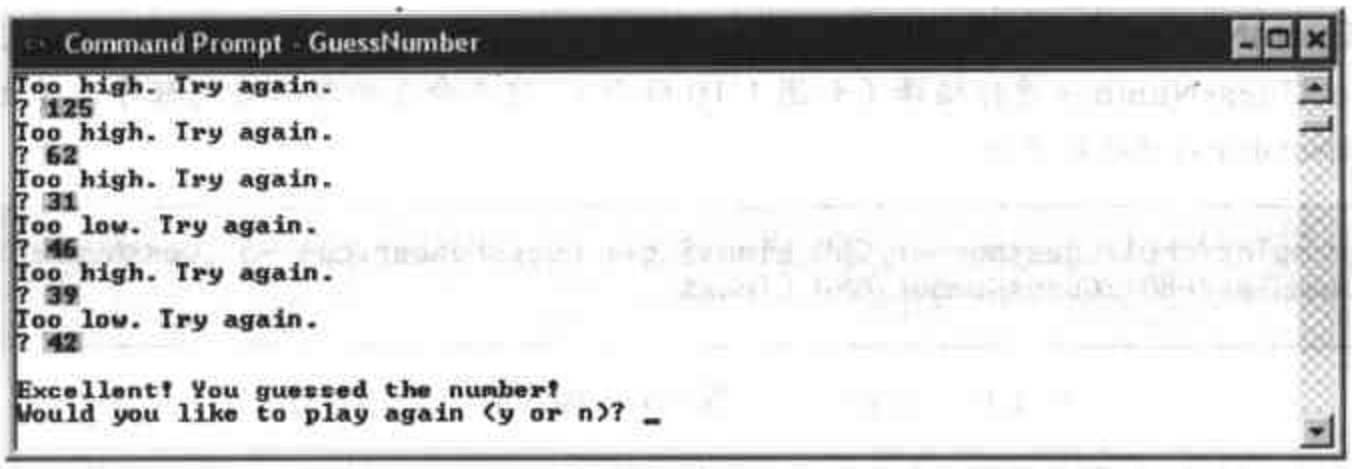


图 1.6　输入其他猜测的数，直到猜对为止

7. **再玩一次游戏或者退出应用程序。**在猜对数字之后，应用程序会问你是否要再玩一次游戏（如图 1.6 所示）。在 “Would you like to play again(y or n)?”（你还要再玩一次吗（是或否）?）的提示下，如果输入字母 y，应用程序就会选择一个新数并显示消息 “Please type your first guess.”，后面跟着问号提示符（如图 1.7 所示）。这样,你就可以在新游戏里进行第一次猜数了；如果输入字母 n，应用程序就会结束，并返回到命令提示符的程序目录下（如图 1.8 所示）。每次你从开始（即步骤 3）执行这个应用程序时，它都会选择相同的数让你猜。
8. **关闭命令提示符窗口。**

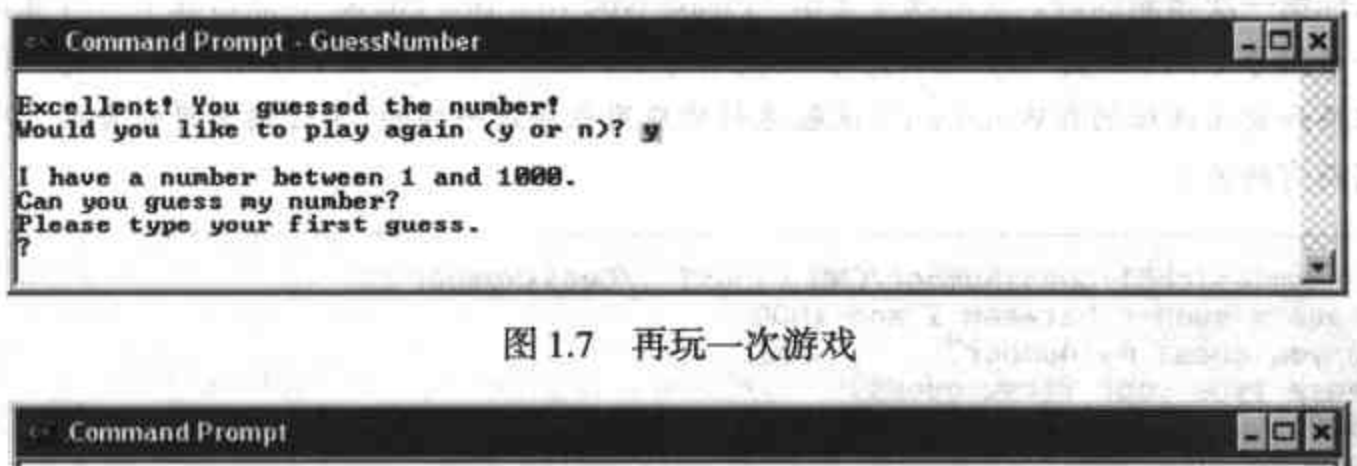


图 1.7　再玩一次游戏

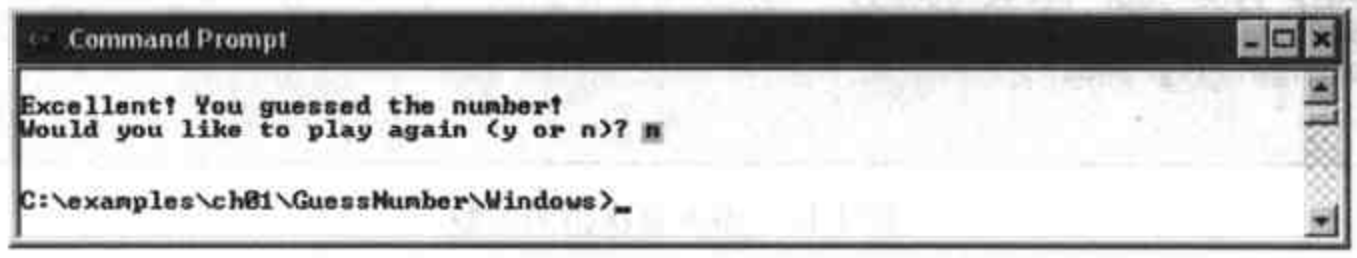


图 1.8　退出游戏

在 Linux 下使用 GNU C++ 运行 C++ 应用程序

在这次试验运行之前，我们假定你是知道如何将例子复制到你的主目录下的。如果对于 Linux 下的文件复制有问题的话，应请教你的指导教师。而且，在本部分的图中，我们会采用粗体来突出每个步骤所要求的用户。我们系统的 shell 提示是用发音符号(~)表示主目录的，每条提示以美元标记符号($)结束。不同的 Linux 系统的提示符也会有所变化。

1. **定位到已完成的应用程序。**在 Linux shell 中，输入 cd Examples\ch01\GuessNumber\GNU_Linux 转到 GuessNumber 应用程序的目录（如图 1.9 所示），然后按回车键。cd 命令是用来改变目录的。

```
~$ cd examples/ch01/GuessNumber/GNU_Linux
~/examples/ch01/GuessNumber/GNU_Linux$
```

图 1.9　登录到你的 Linux 账号以后，转到 GuessNumber 应用程序目录

2. **编译GuessNumber应用程序。**要在GNU C++编译器下运行应用程序，必须首先输入g++ GuessNumber.cpp-o GuessNumber进行编译（如图1.10所示）。这个命令能编译应用程序，并生成一个称为GuessNumber的可执行文件。

```
~/examples/ch01/GuessNumber/GNU_Linux$ g++ GuessNumber.cpp -o GuessNumber
~/examples/ch01/GuessNumber/GNU_Linux$
```

图1.10 用g++命令编译GuessNumber应用程序

3. **运行GuessNumber应用程序。**要想运行可执行文件GuessNumber，在下一个提示下输入./GuessNumber，然后按回车键（如图1.11所示）。

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

图1.11 运行GuessNumber应用程序

4. **输入你第一次猜测的数。**应用程序显示“Please type your first guess.”，然后在下一行显示作为提示符的一个问号（?）（如图1.11所示）。在此提示符下，输入500（如图1.12所示）。注意：这里运行的应用程序和前面改编的在Windows下试验运行的应用程序是一样的，不过输出可能会因为所用编译器的不同而有所差异。

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

图1.12 输入首次猜测的数

5. **输入下一次猜测的数。**应用程序显示“Too high. Try again.”，说明你输入的数比程序所选的正确数要大（如图1.12所示）。在接着的提示下输入250（如图1.13所示）后，应用程序显示“Too low. Try again.”（太小了，再试一次），因为你输入的数比正确答案小。

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too low. Try again.
?
```

图1.13 输入下一次猜测的数并收到反馈

6. **输入其他猜测的数。**输入数继续玩这个游戏（如图1.14所示），直到你猜对为止。一旦猜对了，应用程序会显示“Excellent! You guessed the number!”（如图1.14所示）。

```
Too low. Try again.
? 375
Too low. Try again.
? 437
Too high. Try again.
? 406
Too high. Try again.
? 391
Too high. Try again.
? 383
Too low. Try again.
? 387
Too high. Try again.
? 385
Too high. Try again.
? 384

Excellent! You guessed the number.
Would you like to play again (y or n)?
```

图 1.14　输入其他猜测的数，直到猜对为止

7. **再玩一次游戏或者退出应用程序。**在猜对数字之后，应用程序会问你是否要再玩一次游戏。在“Would you like to play again(y or n)?”的提示下，如果输入字母 y，应用程序就会选择一个新数并显示消息“Please type your first guess.”，后面跟着问号提示符（如图 1.15 所示）。这样,你就可以在新游戏里进行第一次猜数了；如果输入字母 n，应用程序就会结束，并返回到 shell 的程序目录下（如图 1.16 所示）。每次你从开始（即步骤 3）执行这个应用程序时，它都会选择相同的数让你猜。

```
Excellent! You guessed the number.
Would you like to play again (y or n)? y

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

图 1.15　再玩一次游戏

```
Excellent! You guessed the number.
Would you like to play again (y or n)? n

~/examples/ch01/GuessNumber/GNU_Linux$
```

图 1.16　退出游戏

1.17　对象技术和 UML 的介绍

现在我们开始面向对象的介绍，这是一种看待世界和编写计算机程序的自然方法。我们的目标是帮助你形成面向对象的思维方式，并向你介绍统一建模语言（UML，Unified Modeling Language）。UML 是一种让设计面向对象软件系统的人可以用计算机行业标准符号来表示系统的图形语言。

基本的对象技术概念

我们以一些关键术语来开始对面向对象的介绍。观察一下真实的世界，随处都能看到对象（object）——人、动物、植物、汽车、飞机、建筑物和计算机等。人类是以对象的方式进行思维的。电话、房屋、交通灯、微波炉和水冷却器只是我们每天身边常见的一些对象。

下来的一个程序，是从用户那里获取两个数，然后计算它们的和并显示结果。围绕该例的论述将告诉你如何进行各种算术运算，如何将运算结果保存起来以备后用。第5个例子则通过示范两个数的各种比较运算和相应比较结果的消息显示，来介绍判断的基本知识。我们逐行分析每个程序，以便你能轻松地学习C++编程。为了帮助你应用在这里学到的技术，我们在本章的习题中提供了许多编程问题。

2.2　第一个 C++ 程序：打印一行文本

不是程序员的人可能会对 C++ 使用的符号感到陌生。我们现在来看一个简单的程序（如图 2.1 所示），它打印一行文本。这个程序说明了 C++ 语言的几个重要特征。我们详细分析每一行。

第 1 行和第 2 行

```
// Fig. 2.1: fig02_01.cpp
// Text-printing program.
```

都以双斜线“//”开头，表明每一行双斜线符号后面的部分是注释（comment）。程序员插入注释为程序做注解，帮助人们阅读和理解程序。在程序运行时，注释不会导致计算机执行任何操作，它们被C++编译器所忽略，不会生成任何机器语言目标代码。注释“Text-printing program”描述这个程序的目的。以双斜线符号开头的注释在当前行的末尾终止，因此称为单行注释（single-line comment）。注意：C++程序员也可以使用C风格的注释，它可以包含多行，以符号“/*”开始，以符号“*/”结束。

```
1  // Fig. 2.1: fig02_01.cpp
2  // Text-printing program.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome to C++!\n"; // display message
9
10    return 0; // indicate that program ended successfully
11
12 } // end function main
```

```
Welcome to C++!
```

图 2.1　文本打印程序



良好的编程习惯 2.1

每个程序的开始都应该有一个注释，描述这个程序的目的、作者、日期和时间（在本书的程序中，我们没有说明作者、日期和时间，因为这些信息在这里都是没有必要的）。

第3行

```
#include <iostream> // allows program to output data to the screen
```

是一个预处理器指令（preprocessor directive），它是发送给C++预处理器（在1.14节中介绍）的一条消息。以井号“#”开始的行，在程序被编译之前由预处理器来处理。这一行通知预处理器将输入/输出流头文件<iostream>的内容包含到程序中。任何程序，只要使用C++风格的流输入/输出来将数据输出到屏幕或者从键盘输入数据，都必须包含这个文件。稍后我们将会看到，图2.1中的程序将数据输出到屏幕。在第6章中我们会更详细地讨论头文件。

2.3 修改第一个 C++ 程序

这一节用两个例子继续介绍C++编程，说明如何修改图2.1中的程序，从而使用多条语句在一行上打印文本，以及使用一条语句在多行上打印文本。

使用多条语句打印一行文本

打印 Welcome to C++!有好多种方式。例如，图 2.3 在多条语句（第 8~9 行）中使用流插入运算，但是产生的输出与图2.1所示程序的输出完全相同。（注意：从现在开始，我们用比代码部分背景色深的灰色阴影来突出每个程序所介绍的主要特征。）每个流插入都会在上一个流插入停止打印的地方继续打印。第8行的第一个流插入打印Welcome和一个空格，第9行的第二个流插入继续在同一行紧接着空格后开始打印。一般来说，C++ 允许程序员以多种方式表达语句。

```
1  // Fig. 2.3: fig02_03.cpp
2  // Printing a line of text with multiple statements.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome ";
9     std::cout << "to C++!\n";
10
11    return 0; // indicate that program ended successfully
12
13 } // end function main
```

```
Welcome to C++!
```

图 2.3　使用多条语句打印一行文本

使用一条语句打印多行文本

如图2.4中的第8行所示，利用换行符，单独的一条语句也可以打印出多行文本。每次在输出流中遇到\n（换行符）转义序列时，屏幕光标便移到下一行的开头。若想在输出中产生一个空行，可以像第 8 行那样连续使用两个换行符。

```
1  // Fig. 2.4: fig02_04.cpp
2  // Printing multiple lines of text with a single statement.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome\n to\n\n C++!\n";
9
10    return 0; // indicate that program ended successfully
11
12 } // end function main
```

```
Welcome
 to

 C++!
```

图 2.4　使用一条语句打印多行文本

2.4 另一个C++程序：整数相加

我们的下一个程序使用输入流对象std::cin和流提取运算符>>，获取用户通过键盘输入的两个整数，计算它们的和，再用std::cout输出结果。程序、示范的输入和相应的输出如图2.5所示。注意，我们用粗体突出了用户的输入。

```
1  // Fig. 2.5: fig02_05.cpp
2  // Addition program that displays the sum of two numbers.
3  #include <iostream> // allows program to perform input and output
4
5  // function main begins program execution
6  int main()
7  {
8     // variable declarations
9     int number1; // first integer to add
10    int number2; // second integer to add
11    int sum; // sum of number1 and number2
12
13    std::cout << "Enter first integer: "; // prompt user for data
14    std::cin >> number1; // read first integer from user into number1
15
16    std::cout << "Enter second integer: "; // prompt user for data
17    std::cin >> number2; // read second integer from user into number2
18
19    sum = number1 + number2; // add the numbers; store result in sum
20
21    std::cout << "Sum is " << sum << std::endl; // display sum; end line
22
23    return 0; // indicate that program ended successfully
24
25 } // end function main
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

图2.5 显示从键盘输入的两个整数之和的加法程序

第1、2行的注释

```
// Fig. 2.5: fig02_05.cpp
// Addition program that displays the sum of two numbers.
```

陈述了文件的名字和程序的目的。第3行的C++预处理器指令

```
#include <iostream> // allows program to perform input and output
```

将iostream头文件的内容包含到程序中。

程序从第6行的main函数开始执行。第7行的左花括号标记main函数体的开始，第25行相应的右花括号标记main函数的结束。

第9~11行

```
int number1; // first integer to add
int number2; // second integer to add
int sum; // sum of number1 and number2
```

是声明语句。标识符number1、number2和sum是变量名。变量代表计算机内存中的一个特定区域，可以存储程序使用的一个值。这些声明指定变量number1、number2和sum是int类型的数据，这意

在图 2.5 的加法程序中，当执行第 14 行的语句

```
std::cin >> number1; // read first integer from user into number1
```

时，用户输入的数字字符被转换成一个整数，然后它被存放到C++编译器为变量名number1分配的内存区域中。假设用户输入45作为number1的值，那么如图2.6所示，计算机将把45存放到number1的内存区域中。



图 2.6　显示了变量 number1 的名字和值的内存区域

无论什么时候将一个值存放到一块内存区域，它都会覆盖这个区域中原有的值。因此，存放一个新值到一块内存区域的过程是破坏性的。

再回到我们的加法程序，当第 17 行的语句

```
sstd::cin >> number2; // read second integer from user into number2
```

被执行时，假设用户输入值 72，那么这个值被存放到 number2 的内存区域中，内存状况如图 2.7 所示。注意，这两块内存区域在内存中不一定是相邻的。

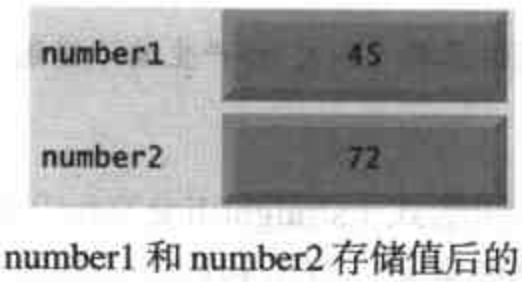


图 2.7　number1 和 number2 存储值后的内存区域

一旦程序获得了 number1 和 number2 的值，就将这两个值相加，并将和存放到变量 sum 中。执行加法的语句

```
sum = number1 + number2; // add the numbers; store result in sum
```

也会将原先存放在 sum 中的值替换掉。这当然发生在 number1 与 number2 的和被存放到 sum 内存区域的那一刻（不管 sum 中已有的是什么值，这个值还是丢失了）。“和”计算之后的内存状况如图2.8所示。请注意，number1 和 number2 的值在求 sum 的前后是不变的。当计算机执行计算时，虽然使用了这两个值，但是没有破坏它们。因此，当从一个内存区域读取出一个值时，这个过程是非破坏性的。

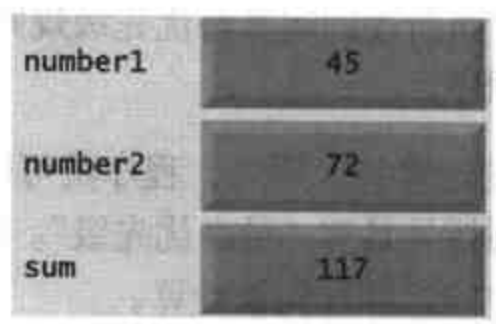


图 2.8　计算并存储 number1 和 number2 的和到 sum 之后的内存区域

2.6　算术运算

大多数程序会执行算术运算。图 2.9 总结了 C++ 的算术运算符。注意，这里使用了一些在代数中不使用的特殊符号。星号（*）表示乘法，百分号（%）是求模运算符，稍后会做讨论。图 2.9 中

运算符优先级规则集合规定了 C++ 应用运算符的次序。当我们说某些运算符从左到右被计算时，是指运算符的结合律。例如，在表达式 a+b+c 中，加法运算符（+）按从左到右的顺序进行结合，所以先计算 a+b，然后 c 再加入到它们的和中得到整个表达式的值。我们将会看到有些运算符是按从右到左的顺序结合的。图 2.10 总结了运算符优先级的规则。引入新的 C++ 运算符时，这个表还可以继续扩充。附录 A 中包含了完整的优先级表。

运算符	运算	求值顺序（优先级）
()	圆括号	最先求值。如果圆括号是嵌套的，那么先求最内层圆括号内的表达式；如果“在同一层次上”（即非嵌套的）有多对圆括号，按从左到右的顺序求值
*	乘法	其次求值。如果同时有多个运算符出现，按从左到右的顺序求值
/	除法	
%	求模	
+	加法	最后求值。如果同时有多个运算符出现，按从左到右的顺序求值
-	减法	

图 2.10　算术运算符的优先级

代数表达式和 C++ 表达式示例

根据运算符优先级的规则，现在来看几个表达式。每个例子列出一个代数表达式和与它等价的 C++ 表达式。下面是计算 5 个数的算术平均值的例子：

代数表达式：$m = \dfrac{a+b+c+d+e}{5}$

C++ 表达式：

```
m = ( a + b + c + d + e ) / 5;
```

这里必须加圆括号，因为除法的优先级比加法的高。是整个(a+b+c+d+e)的值要被 5 来除，所以，如果圆括号被错误地省略掉的话，我们得到的是 a+b+c+d+e/5，它会被当做

$$a+b+c+d+\frac{e}{5}$$

而导致了错误的计算。下面是直线方程的例子：

代数表达式：$y = mx + b$

C++ 表达式：

```
y = m * x + b;
```

在此不需要圆括号。因为乘法的优先级比加法高，所以先计算乘法。

接下来的例子包含了求模（%）、乘法、除法、加法、减法和赋值运算：

代数表达式：$z = pr\%q + w/x - y$

C++ 表达式：

```
z = p * r % q + w / x - y;
```

6　1　2　4　3　5

这里语句下圆圈内的数字指出了 C++ 应用运算符的次序。因为乘法、求模和除法的优先级比加法和减法的高，所以它们按从左到右的顺序（即它们是从左到右结合的）先被计算。然后计算加法和减法，也是按从左到右的顺序进行。最后进行赋值运算。

二次多项式的计算

为了更好地理解运算符优先级的规则，我们来看看二次多项式 $y = ax^2 + bx + c$ 的计算：

```
y = a * x * x + b * x + c;
```

6 1 2 4 3 5

这里语句下面的数字指出了C++应用运算符的次序。在C++中，没有求幂的算术运算符，因此我们把x^2表示为x*x。很快，我们将讨论用来求幂的标准库函数pow（"power"）。因为还有一些关系到pow所要求数据类型的小问题需要解决，所以我们将pow的详细解释推迟到第6章。



常见的编程错误 2.4

一些程序设计语言使用运算符**或^表示求幂运算。C++不支持这些求幂运算符，所以使用它们会产生错误。

假设在上述的二次多项式中，变量a、b、c和x初始化如下：a=2，b=3，c=7，x=5。图2.11说明了运算符应用的次序。

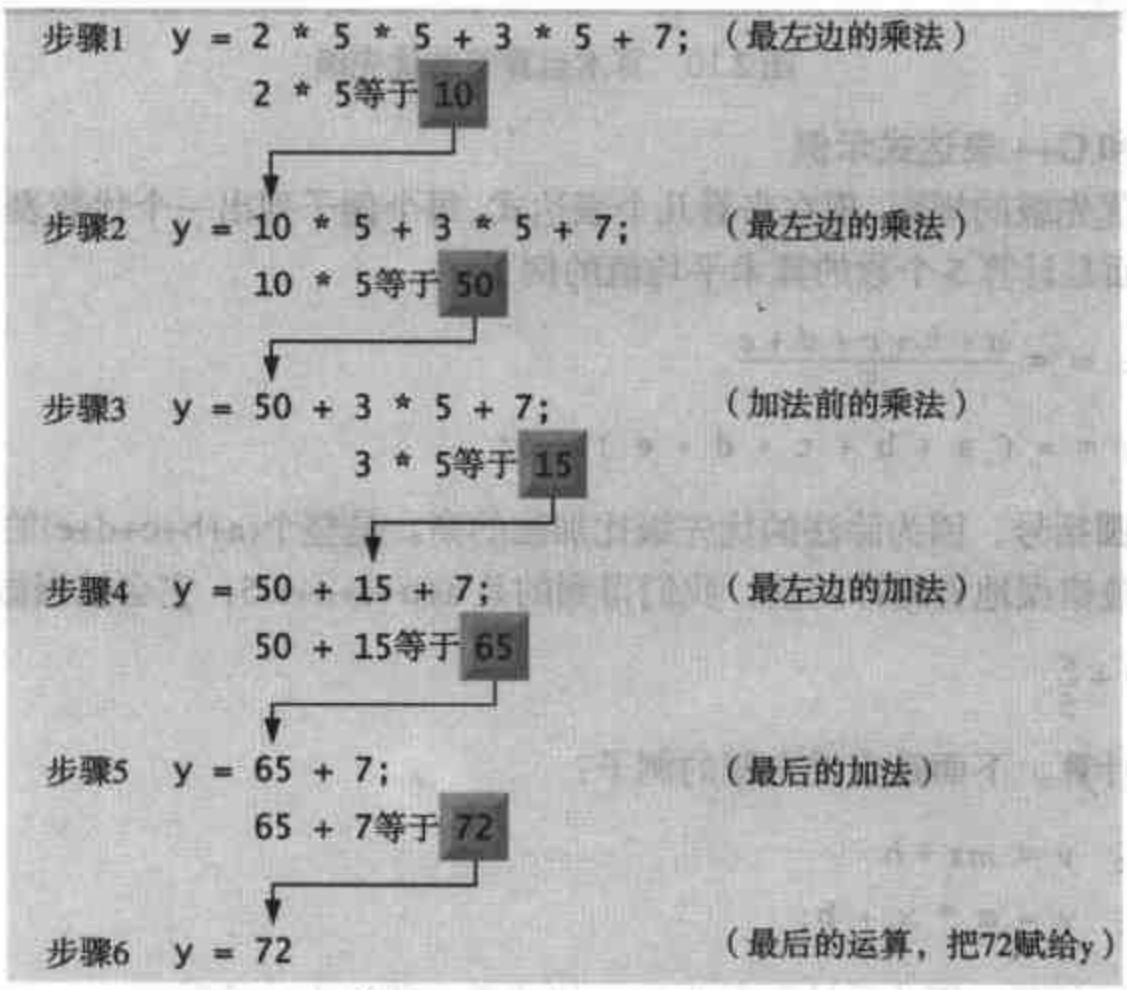


图2.11 一个二次多项式的计算顺序

和在代数中一样，为了使表达式更清楚，允许在表达式中放置一些非必需的圆括号。这些圆括号称为冗余的圆括号。例如，前面的赋值语句可按下面的方式加圆括号：

```
y = ( a * x * x ) + ( b * x ) + c;
```



良好的编程习惯 2.14

在复杂的算术表达式中使用冗余的圆括号，可以使表达式更加清晰。

2.7 判断：相等运算符和关系运算符

这一节介绍C++的if语句的一个简单版本。if语句可以使程序根据某些条件的真或假做出判定。如果满足条件，即条件是真的，在if语句体中的语句就会被执行；如果不满足条件，即条件是假的，if语句体中的语句就不会被执行。我们马上会用相应的例子进行说明。

```
1  // Fig. 2.13: fig02_13.cpp
2  // Comparing integers using if statements, relational operators
3  // and equality operators.
4  #include <iostream> // allows program to perform input and output
5
6  using std::cout; // program uses cout
7  using std::cin;  // program uses cin
8  using std::endl; // program uses endl
9
10 // function main begins program execution
11 int main()
12 {
13    int number1; // first integer to compare
14    int number2; // second integer to compare
15
16    cout << "Enter two integers to compare: "; // prompt user for data
17    cin >> number1 >> number2; // read two integers from user
18
19    if ( number1 == number2 )
20       cout << number1 << " == " << number2 << endl;
21
22    if ( number1 != number2 )
23       cout << number1 << " != " << number2 << endl;
24
25    if ( number1 < number2 )
26       cout << number1 << " < " << number2 << endl;
27
28    if ( number1 > number2 )
29       cout << number1 << " > " << number2 << endl;
30
31    if ( number1 <= number2 )
32       cout << number1 << " <= " << number2 << endl;
33
34    if ( number1 >= number2 )
35       cout << number1 << " >= " << number2 << endl;
36
37    return 0; // indicate that program ended successfully
38
39 } // end function main
```

```
Enter two integers to compare: 3 7
3 != 7
3 < 7
3 <= 7
```

```
Enter two integers to compare: 22 12
22 != 12
22 > 12
22 >= 12
```

```
Enter two integers to compare: 7 7
7 == 7
7 <= 7
7 >= 7
```

图 2.13　相等运算符和关系运算符

f) cout << "The product is " << result << endl;

g) return 0;

2.5

```
1  // Calculate the product of three integers
2  #include <iostream> // allows program to perform input and output
3
4  using std::cout; // program uses cout
5  using std::cin; // program uses cin
6  using std::endl; // program uses endl
7
8  // function main begins program execution
9  int main()
10 {
11    int x; // first integer to multiply
12    int y; // second integer to multiply
13    int z; // third integer to multiply
14    int result; // the product of the three integers
15
16    cout << "Enter three integers: "; // prompt user for data
17    cin >> x >> y >> z; // read three integers from user
18    result = x * y * z; // multiply the three integers; store result
19    cout << "The product is " << result << endl; // print result; end line
20
21    return 0; // indicate program executed successfully
22 } // end function main
```

2.6　a) 错误：在 if 语句中，条件的右圆括号后面多了个分号。

改正：删去右圆括号后面的分号。注意：这个错误会导致，不管 if 语句中条件是否为真，输出语句都会被执行。右圆括号后面的分号是一条空语句，即一条不做任何事情的语句。我们在下一章将会进一步学习空语句。

b) 错误：关系运算 =>

改正：把 =>改正为>=。同时也可以把“equal to or greater than”改为“greater than or equal to”。

习题

2.7　请论述下面每个对象的含义：

a) std::cin

b) std::cout

2.8　填空题

a) ____ 用于为程序做注解，提高程序的可读性。

b) 用来在屏幕上打印信息的对象是 ____。

c) 一条用于做出判断的 C++ 语句是 ____。

d) 大多数的计算通常由 ____ 语句执行。

e) ____ 对象通过键盘输入值。

2.9　编写单条 C++ 语句，完成下列各项任务。

a) 打印信息 "Enter two numbers"。

b) 将变量 b 和 c 的乘积赋给变量 a。

c) 说明程序执行工资的计算（即使用帮助注解程序的文本）。

```
1  // Fig. 3.1: fig03_01.cpp
2  // Define class GradeBook with a member function displayMessage;
3  // Create a GradeBook object and call its displayMessage function.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    // function that displays a welcome message to the GradeBook user
13    void displayMessage()
14    {
15       cout << "Welcome to the Grade Book!" << endl;
16    } // end function displayMessage
17 }; // end class GradeBook
18
19 // function main begins program execution
20 int main()
21 {
22    GradeBook myGradeBook; // create a GradeBook object named myGradeBook
23    myGradeBook.displayMessage(); // call object's displayMessage function
24    return 0; // indicate successful termination
25 } // end main
```

```
Welcome to the Grade Book!
```

图 3.1　定义具有一个成员函数的 GradeBook 类，创建一个 GradeBook 对象并调用它的成员函数

我们首先描述如何定义类和成员函数，然后解释如何创建对象以及如何调用对象的成员函数。开始的几个例子在同一个文件中包含了 main 函数和它使用的 GradeBook 类。在本章稍后部分，我们将介绍可以较好实践软件工程的更复杂的构造程序的方法。

GradeBook 类

在第 20~25 行中的 main 函数可以创建 GradeBook 类的一个对象前，我们必须告诉编译器属于该类的成员函数和数据成员是什么。这就是人们熟知的定义一个类。GradeBook 的类定义（第 9~17 行）包含一个名为 displayMessage 的成员函数（第 13~16 行），它在屏幕上显示一条信息（第 15 行）。我们曾说过，一个类就像一张蓝图，因此我们需要产生 GradeBook 类的一个对象（第 22 行），调用它的成员函数 displayMessage（第 23 行）让第 15 行执行并显示欢迎信息。我们很快将详细地解释第 22~23 行。

第 9 行以关键字 class 开始了 GradeBook 类的定义。关键字 class 后跟类名 GradeBook。按照惯例，用户定义的类的名字以大写字母开头，而且为了增强可读性，类名中每个随后的单词其首字母也为大写。这种大写风格常称为骆驼风格（camel case），因为大写和小写字母相混合形成的这种图案颇似骆驼的轮廓。

如第 10 和 17 行所示，每个类的体（body）包围在一对花括号中（{和}）。类的定义以分号结束（第 17 行）。



常见的编程错误 3.1

在类定义的末尾忘记分号是一个语法错误。

可以用来创建对象。程序员可以定义所需的新类类型。这正是为什么会认为C++是一种可扩展语言的原因之一。

在第23行中，通过在变量名myGradeBook后依次加点运算符（.）、函数名displayMessage和空的圆括号对的方式，调用了定义在第13~16行中的成员函数displayMessage。此次调用使displayMessage函数执行它的任务。在第23行的起始部分，“myGradeBook.”指明main函数应当使用第22行创建的这个GradeBook对象。第13行中空的圆括号表明成员函数displayMessage执行它的任务不需要额外的数据（在3.5节，将会看到怎样向函数传递数据）。当成员函数displayMessage完成它的任务后，main函数继续执行第24行。该行表示main顺利完成了它的任务，这是main的结尾，所以程序终止。

GradeBook 类的 UML 类图

如1.17节中提到的，UML是一种图形表示语言，程序员用它以一种标准化的形式描述他们的面向对象系统。在UML中，每个类在类图中表示为一个由三部分组成的矩形。图3.2呈现了图3.1中类GraqdeBook的UML类图。这种矩形的上部包含水平居中、黑体的类的名字；中部包含与C++中数据成员相对应的类的属性。在图3.2中，矩形中部是空白的，因为图3.1中GradeBook类的这一版本没有任何属性（3.6节给出的GradeBook类的版本具有一个属性）。矩形的下部包含类的操作，这对应C++中的成员函数。UML通过操作名后接圆括号对的形式来表示操作。GradeBook类只有一个名为displayMessage的成员函数，所以在图3.2的下部列出了一个以此为名的操作。成员函数displayMessage执行其任务时不需要额外的数据，因此同图3.1中第13行的成员函数displayMessage的头部一样，类图中displayMessage之后的圆括号内是空的。操作名之前的加号（+）表明在UML中displayMessage是一个公共操作（即在C++中是一个public成员函数）。我们常常使用UML类图概括类的属性和操作。

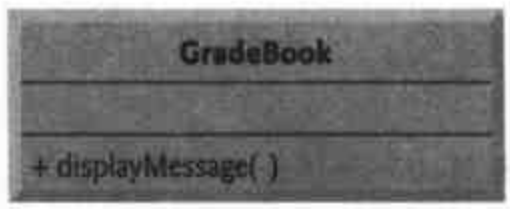


图3.2 UML类图指出GradeBook类具有一个公共的displayMessage操作

3.5 定义具有形参的成员函数

在3.2节的小轿车类比中，曾提到踩下小轿车的油门踏板意味着向小轿车发送一条执行任务的消息，即令小轿车加速的消息。可是小轿车应当加速到多快呢？大家都知道，踏板踩得越低，小轿车速度越快。因此，发送到小轿车的消息包括了要执行的任务和附加的帮助小轿车执行任务的信息。这一附加的信息就称为形参（parameter）——形参的值帮助小轿车决定了加速的程度。类似地，一个成员函数可以要求一个或多个形参，用于描述它执行任务所需要的额外数据。对于函数的每一个形参，每次函数调用会给它们提供值，这些值称为实参（argument）。例如，要向银行账户存一笔钱，假定Account类的deposit成员函数指定了表示存钱数目的一个形参。当deposit成员函数被调用时，表示存钱数目的实参值被复制到该成员函数的形参中，然后该成员函数将此数目加入到账户余额。

定义和测试 GradeBook 类

下一个例子（如图3.3所示）重新定义了GradeBook类（第14~23行），它的成员函数displayMessage（第18~22行）将课程名称作为欢迎消息的一部分。这个新的成员函数displayMessage规定了一个表示要输出的课程名称的形参（即第18行的courseName）。

```
1  // Fig. 3.3: fig03_03.cpp
2  // Define class GradeBook with a member function that takes a parameter;
3  // Create a GradeBook object and call its displayMessage function.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include <string> // program uses C++ standard string class
10 using std::string;
11 using std::getline;
12
13 // GradeBook class definition
14 class GradeBook
15 {
16 public:
17    // function that displays a welcome message to the GradeBook user
18    void displayMessage( string courseName )
19    {
20       cout << "Welcome to the grade book for\n" << courseName << "!"
21          << endl;
22    } // end function displayMessage
23 }; // end class GradeBook
24
25 // function main begins program execution
26 int main()
27 {
28    string nameOfCourse; // string of characters to store the course name
29    GradeBook myGradeBook; // create a GradeBook object named myGradeBook
30
31    // prompt for and input course name
32    cout << "Please enter the course name:" << endl;
33    getline( cin, nameOfCourse ); // read a course name with blanks
34    cout << endl; // output a blank line
35
36    // call myGradeBook's displayMessage function
37    // and pass nameOfCourse as an argument
38    myGradeBook.displayMessage( nameOfCourse );
39    return 0; // indicate successful termination
40 } // end main
```

```
Please enter the course name:
CS101 Introduction to C++ Programming

Welcome to the grade book for
CS101 Introduction to C++ Programming!
```

图3.3　定义具有一个成员函数的GradeBook类，该成员函数接受一个形参

在讨论GradeBook类的新特征之前，我们先了解一下第26~40行定义的main函数是如何使用这个新类的。第28行创建了一个类型为string、名为nameOfCourse的变量，用于存储用户输入的课程名称。类型为string的变量表示一串字符，比如“CS101 Introduction to C++ Programming”。一个字符串实际上是C++标准库string类的一个对象。这个类在头文件<string>中定义，并且string这个名字和cout一样属于命名空间std。为了使第28行通过编译，第9行包含了<string>头文件。请注意，通过第10行中的using声明，我们就可以在第28行中简单地写string而不是std::string。现在，我们可以像考虑int等其他类型的变量一样考虑string变量。你将在3.10节学习到string的其他性能。



良好的编程习惯 3.1
为了避免二义性，最好不要使传递给函数的实参与函数定义中相应的形参同名。



良好的编程习惯 3.2
选择有明确含义的函数名称和形参名称可以使程序更具有可读性，而且还有助于避免使用大量的注释。

更新的 GradeBook 类的 UML 类图

图 3.4 是图 3.3 中 GradeBook 类的 UML 类图表示。如同图 3.1 中所定义的 GradeBook 类一样，这个 GradeBook 类还是包含了 public 的成员函数 displayMessage。但是，这一版本的 displayMessage 带有一个形参。UML 表示一个形参的方式是在操作名后的圆括号内依次列出形参名、冒号和参数的类型。UML 具有与 C++ 类似的数据类型。UML 是与语言无关的——它和许多不同的程序设计语言配合使用，所以，它的术语与 C++ 的并不完全一样。例如，UML 的类型 String 与 C++ 的类型 string 对应。GradeBook 类的成员函数 displayMessage（如图 3.3 中的第 18~22 行所示）有一个名为 courseName 的 string 形参，因此图 3.4 在操作名 displayMessage 后的圆括号内列出了 courseName: String。请注意，GradeBook 类的这个版本仍然没有任何数据成员。

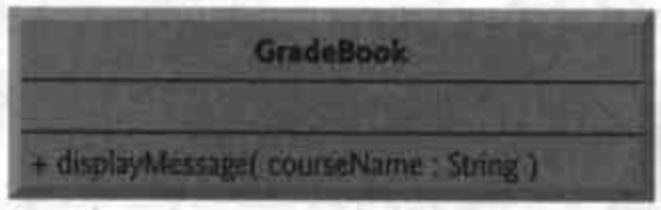


图 3.4　UML 类图指出 GradeBook 类有一个操作 displayMessage，该操作具有一个 UML 类型 String 的形参 courseName

3.6　数据成员、设置函数和获取函数

在第 2 章中，我们在 main 函数中声明了程序的所有变量。声明在一个函数定义的体中的变量被认为是局部变量，只能在函数中声明它们的行到函数定义结束的右花括号（}）之间使用。函数中的局部变量必须在使用前声明，并且不能在声明它们的函数外部被访问。当函数结束时，其局部变量的值将丢失（当然，在第 6 章讨论 static 局部变量时，你会看到例外）。回顾 3.2 节的内容，对象在程序中使用时，它具有自己的属性。这样的属性存在于对象的整个生命周期中。

类通常由一个或多个成员函数组成，这些成员函数操纵属于该类的某个特定对象的属性。在类的定义中属性表示为变量。这样的变量称为数据成员，声明在类的定义之内，但是在所有类的成员函数的定义体之外。类的每个对象管理它自己的在内存中的一份属性副本。本节中的例子展示了包含一个数据成员 courseName 的 GradeBook 类，courseName 代表了某个特定的 GradeBook 对象的课程名称。

具有一个数据成员、一个设置函数和一个获取函数的 GradeBook 类

在下面的例子中，GradeBook 类（如图 3.5 所示）将课程名称作为一个数据成员维护，所以课程名称可以在程序运行过程的任何时候使用或被修改。该类包含了 setCourseName、getCourseName 和 displayMessage 三个成员函数。成员函数 setCourseBook 将课程名称存放到 GradeBook 的数据成员中，而成员函数 getCourseName 则从数据成员那里获取 GradeBook 的课程名称。成员函数 displayMessage 目前没有指定任何形参，仍然是显示一条包含了课程名称的欢迎信息。但是，正如所看到的，现在该函数是通过调用同一个类中的另一个函数 getCourseName 来获得课程名称的。

```
1  // Fig. 3.5: fig03_05.cpp
2  // Define class GradeBook that contains a courseName data member
3  // and member functions to set and get its value;
4  // Create and manipulate a GradeBook object with these functions.
5  #include <iostream>
6  using std::cout;
7  using std::cin;
8  using std::endl;
9
10 #include <string> // program uses C++ standard string class
11 using std::string;
12 using std::getline;
13
14 // GradeBook class definition
15 class GradeBook
16 {
17 public:
18    // function that sets the course name
19    void setCourseName( string name )
20    {
21       courseName = name; // store the course name in the object
22    } // end function setCourseName
23
24    // function that gets the course name
25    string getCourseName()
26    {
27       return courseName; // return the object's courseName
28    } // end function getCourseName
29
30    // function that displays a welcome message
31    void displayMessage()
32    {
33       // this statement calls getCourseName to get the
34       // name of the course this GradeBook represents
35       cout << "Welcome to the grade book for\n" << getCourseName() << "!"
36          << endl;
37    } // end function displayMessage
38 private:
39    string courseName; // course name for this GradeBook
40 }; // end class GradeBook
41
42 // function main begins program execution
43 int main()
44 {
45    string nameOfCourse; // string of characters to store the course name
46    GradeBook myGradeBook; // create a GradeBook object named myGradeBook
47
48    // display initial value of courseName
49    cout << "Initial course name is: " << myGradeBook.getCourseName()
50       << endl;
51
52    // prompt for, input and set course name
53    cout << "\nPlease enter the course name:" << endl;
54    getline( cin, nameOfCourse ); // read a course name with blanks
55    myGradeBook.setCourseName( nameOfCourse ); // set the course name
56
57    cout << endl; // outputs a blank line
58    myGradeBook.displayMessage(); // display message with new course name
```

图 3.5　定义和测试具有一个数据成员以及设置和获取函数的 GradeBook 类

```
59      return 0; // indicate successful termination
60   } // end main
```

```
Initial course name is:

Please enter the course name:
CS101 Introduction to C++ Programming

Welcome to the grade book for
CS101 Introduction to C++ Programming!
```

图3.5　定义和测试具有一个数据成员以及设置和获取函数的GradeBook类（续）



良好的编程习惯3.3
在多个成员函数的定义之间加入空行，可以增强程序的可读性。

通常教师往往讲授不止一门课程，每门课都有它自己的课程名称。第39行声明了一个名为courseName的string类型的变量。该声明处在类的定义（第15~40行）之内，但是在所有类的成员函数的定义体（第19~22、25~28、31~37行）之外，所以第39行是对一个数据成员的声明。GradeBook类的每个实例（也就是对象）包含了类的每个数据成员的一份副本。例如，如果有GradeBook类的两个对象，那么如图3.7的例子所示，每个对象各自具有courseName的一份副本。将courseName设计为数据成员的一个好处就是：类（在本例中为GradeBook）的所有成员函数都可以操纵类定义中出现的任何数据成员（在本例中为courseName）。

成员访问说明符public和private

绝大多数数据成员的声明出现在成员访问说明符标志private:（第38行）之后。和public一样，关键字private是一个成员访问说明符。在成员访问说明符private之后（而且在下一个成员访问说明符之前）声明的变量或者函数，只可以被声明它们的类的成员函数所访问。因此，数据成员courseName只能够用在GradeBook类的（每个对象的）成员函数setCourseName、getCourseName和displayMessage中。数据成员courseName，由于它是私有的，所以不能被类之外的函数（例如main）或程序中其他类的成员函数访问。在程序中，试图通过诸如myGradeBook.courseName之类的表达式访问数据成员courseName，将导致编译错误，该错误包含类似下面的一条信息：

```
cannot access private member declared in class 'GradeBook'
```



软件工程知识3.1
根据经验，数据成员应该声明为私有（private）的，成员函数应该声明为公有（public）的。我们将看到，如果某些成员函数只是被该类的其他成员函数访问，那么它们更适合声明为私有的。



常见的编程错误3.6
不是某个特定类的成员（或者友元，将在第10章学习）的函数，试图访问该类的private成员时会产生一个编译错误。

类成员的默认成员访问说明符是private，因此类头部之后、第一个成员访问说明符之前的所有成员都是私有的。成员访问说明符public和private可以重复使用，但是这是不必要的，而且容易造成混淆。



良好的编程习惯3.4
尽管事实上public和private成员访问说明符可以重复和混合地使用，但是将类的所有public成员列在一个组内，而将类的所有private成员列在另一个组内，这样更容易使客户的注意力集中在类的公共接口上，而不是在类的实现上。

中的返回类型 String。请注意，操作 setCouseName 和 displayMessage 没有返回值，即它们返回 void，所以 UML 类图在这些操作的圆括号后没有指定返回类型。当函数没有返回值时，UML 并不采用 C++ 所使用的 void。

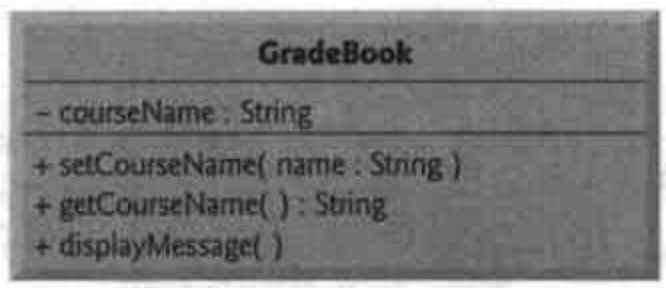


图 3.6　GradeBook 类的 UML 类图，包含一个 private 属性 courseName 及 public 操作 setCourseName、getCourseName 和 displayMessage

3.7　用构造函数初始化对象

我们在 3.6 节中曾提到，在创建 GradeBook 类的一个对象（如图 3.5 所示）时，该对象的数据成员 courseName 被初始化为默认的空串。那么，如果想在创建一个 GradeBook 对象的同时就提供课程名称的话，该怎么办呢？实际上，你声明的每个类都可以提供一个构造函数（constructor），用于类对象创建时的初始化。构造函数是一种特殊的成员函数，定义时必须和类同名，这样编译器才能够将它和类的其他成员函数区分开来。构造函数和其他函数之间的一个重大差别是构造函数不能返回值，因此对它们不可以指定返回类型（甚至连 void 都不行）。通常情况下，构造函数声明为 public。术语“constructor”常常在文献中缩写为“ctor”——我们更喜欢不使用缩写。

对于每个被创建的对象，C++ 要求一次构造函数的调用，这有助于保证在程序使用该对象前，它被正确初始化。构造函数的调用隐式地发生在对象创建时。在任何没有显式地包含构造函数的类中，编译器会提供一个默认的构造函数，更确切地说，一个没有形参的构造函数。例如，当图 3.5 中的第 46 行创建一个 GradeBook 的对象时，这个默认的构造函数就被调用了，因为 myGradeBook 的声明没有明确说明任何的构造函数的实参。由编译器提供的默认的构造函数创建GradeBook的对象时，没有给对象的数据成员任何初始值。注意：对于是其他类的对象的数据成员，默认的构造函数将隐式地调用每个数据成员的默认构造函数，保证数据成员正确初始化。事实上，这正是图 3.5 中 string 数据成员 courseName 被初始化为空串的原因——类 string 的默认的构造函数设置这个串的值为空串。在 10.3 节中，大家将学习更多的有关初始化数据成员的内容，其中的数据成员是其他类的对象。

在图 3.7 的例子中，当创建一个 GradeBook 对象时（第 49 行），我们对它指定课程名称。在这种情况下，实参“CS101 Introduction to C++ Programming”传递给这个 GradeBook 对象的构造函数（第 17~20 行）并用于初始化 courseName。图 3.7 定义了一个修改后的 GradeBook 类，它包含了一个构造函数并且该构造函数具有一个接收初始课程名称的 string 形参。

定义构造函数

图 3.7 中的第 17~20 行定义了 GradeBook 类的一个构造函数。注意，这个构造函数与它的类同名，都称为 GradeBook。构造函数在它的形参列表中指定它执行任务所需要的数据。如第 49~50 行所示，当创建一个新的对象时，在对象名之后的圆括号中放置这些数据。第 17 行指出 GradeBook 类的构造函数有一个名为 name 的 string 形参。注意，第 17 行没有指定返回类型，因为构造函数不可以返回任何值（甚至是 void）。

```
// Fig. 3.7: fig03_07.cpp
// Instantiating multiple objects of the GradeBook class and using
// the GradeBook constructor to specify the course name
// when each GradeBook object is created.
#include <iostream>
using std::cout;
using std::endl;

#include <string> // program uses C++ standard string class
using std::string;

// GradeBook class definition
class GradeBook
{
public:
   // constructor initializes courseName with string supplied as argument
   GradeBook( string name )
   {
      setCourseName( name ); // call set function to initialize courseName
   } // end GradeBook constructor

   // function to set the course name
   void setCourseName( string name )
   {
      courseName = name; // store the course name in the object
   } // end function setCourseName

   // function to get the course name
   string getCourseName()
   {
      return courseName; // return object's courseName
   } // end function getCourseName

   // display a welcome message to the GradeBook user
   void displayMessage()
   {
      // call getCourseName to get the courseName
      cout << "Welcome to the grade book for\n" << getCourseName()
         << "!" << endl;
   } // end function displayMessage
private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook

// function main begins program execution
int main()
{
   // create two GradeBook objects
   GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
   GradeBook gradeBook2( "CS102 Data Structures in C++" );

   // display initial value of courseName for each GradeBook
   cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
      << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
      << endl;
   return 0; // indicate successful termination
} // end main
```

```
gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++
```

图 3.7　实例化 GradeBook 类的多个对象并在创建每个 GradeBook 对象时使用 GradeBook 构造函数指定课程名称

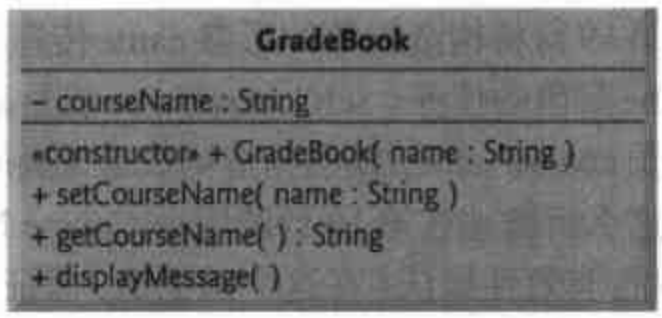


图 3.8 UML 类图指出 GradeBook 类有一个构造函数，该函数具有一个 UML 类型 String 的形参 name

3.8 一个类对应一个独立文件的可重用性

从程序员的角度看，迄今为止我们已经开发了所需的GradeBook类。下面我们将考虑一些软件工程方面的问题解决办法。创建类定义的好处之一是：当正确封装时，类可能被全球范围的程序员所重用。例如，通过在程序中包含头文件<string>（以及正如我们看到的，能够链接到库的对象代码），我们可以在任何一个 C++ 程序中重用 C++ 标准库类型 string。

遗憾的是，希望使用我们的GradeBook类的程序员却不能在另一个程序中简单地包含图3.7的文件。正如在第 2 章所学的那样，main 函数是每个程序执行的起点，每个程序有且仅有一个 main 函数。如果其他程序员包含图 3.7 的文件，他们将得到“超重”的行李，就因为我们的 main 函数。于是，他们的程序将有两个main函数。当他们试图编译这样的程序时，编译器会报告一个错误，因为每个程序只能有一个 main 函数。例如，在 Microsoft Visual C++ .NET 中试图编译具有两个 main 函数的程序时，它产生如下错误信息：

```
error C2084: function 'int main(void)' already has a body
```

（错误 C2084: 函数 'int main(void)' 已经有一个函数体了）

当 GNU C++ 编译器试着编译它遇到的第二个 main 函数时，同样，它产生错误信息：

```
redefinition of 'int main()'
```

这些错误指示一个程序已经有了一个main函数，所以如果在具有一个类定义的相同文件中放置 main 的话，将妨碍这个类被其他程序的重用。在本节中，我们演示使 GradeBook 类可重用的方法，即将 GradeBook 类放到另一个与 main 函数分开的文件中。

头文件

这一章中前面的每个例子都由包含了 GradeBook 的类定义和一个 main 函数的单个.cpp 文件（也称为一个源代码文件）构成。在建立一个面向对象的 C++ 程序时，通常在一个文件中定义可重用源代码（例如，一个类），按照惯例这个文件扩展名为.h ——称为头文件。程序使用 #include 预处理指令包含头文件，并利用可重用软件组件，例如 C++ 标准库提供的类型 string，以及用户自定义的类型如 GradeBook 类等。

在下面的例子中，我们将图 3.7 的代码分成两个文件—— GradeBook.h（如图 3.9 所示）和 fig03_10.cpp（如图 3.10 所示）。在阅读图 3.9 中的头文件时，请注意它仅包含 GradeBook 的类定义（第 11~41 行）和第 3~8 行，其中的第 3~8 行允许 GradeBook 类使用 cout、endl 和类型 string。使用 GradeBook 类的 main 函数定义在源代码文件 fig03_10.cpp（见图 3.10）的第 10~21 行。为了帮助大家做好应对（本书其余部分或今后的职业生涯中遇到的）更大程序的准备，我们常常使用一个包含了 main 函数的独立的源代码文件测试我们的类，这就是所谓的驱动程序。你很快将学习具有 main 函数的源代码文件是如何使用头文件中的类定义创建类的对象的。

包含有一个用户定义类的头文件

头文件，如GradeBook.h（见图3.9），不能用于开始程序的执行，因为它没有包含main函数。如果试图单独编译和连接GradeBook.h，创建一个可执行的应用程序，那么Microsoft Visual C++ .NET将产生一条连接错误信息：

```
error LNK2019: unresolved external symbol _main referenced in
function _mainCRTStartup
```

```
1  // Fig. 3.9: GradeBook.h
2  // GradeBook class definition in a separate file from main.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string> // class GradeBook uses C++ standard string class
8  using std::string;
9
10 // GradeBook class definition
11 class GradeBook
12 {
13 public:
14    // constructor initializes courseName with string supplied as argument
15    GradeBook( string name )
16    {
17       setCourseName( name ); // call set function to initialize courseName
18    } // end GradeBook constructor
19
20    // function to set the course name
21    void setCourseName( string name )
22    {
23       courseName = name; // store the course name in the object
24    } // end function setCourseName
25
26    // function to get the course name
27    string getCourseName()
28    {
29       return courseName; // return object's courseName
30    } // end function getCourseName
31
32    // display a welcome message to the GradeBook user
33    void displayMessage()
34    {
35       // call getCourseName to get the courseName
36       cout << "Welcome to the grade book for\n" << getCourseName()
37          << "!" << endl;
38    } // end function displayMessage
39 private:
40    string courseName; // course name for this GradeBook
41 }; // end class GradeBook
```

图3.9　GradeBook的类定义

在Linux下运行GNU C++产生的一条连接错误信息中包含：

```
undefined reference to 'main'
```

这个错误表明连接器未能找到程序的main函数。要测试定义在图3.9中的GradeBook类，就必须书写包含一个main函数的独立的源代码文件（例如图3.10），它实例化和使用这个类的对象。

```
1  // Fig. 3.10: fig03_10.cpp
2  // Including class GradeBook from file GradeBook.h for use in main.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "GradeBook.h" // include definition of class GradeBook
8
9  // function main begins program execution
10 int main()
11 {
12    // create two GradeBook objects
13    GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
14    GradeBook gradeBook2( "CS102 Data Structures in C++" );
15
16    // display initial value of courseName for each GradeBook
17    cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
18       << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
19       << endl;
20    return 0; // indicate successful termination
21 } // end main
```

```
gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++
```

图3.10 包含源自文件GradeBook.h的GradeBook类以让main函数使用

回想一下在3.4节中，虽然编译器知道基本的数据类型，如int是什么，但是它不知道何谓GradeBook，因为它是一个用户定义的类型。事实上，编译器甚至不知道在C++标准库中的类。为了帮助编译器明白如何使用一个类，我们必须明确地给它提供这个类的定义——这就是为什么使用类型string时，程序必须包含头<string>文件的原因。这使编译器能够确定为这个类的每个对象保留的内存大小，以及能够保证程序正确地调用这个类的成员函数。

为了在图3.10的第13~14行中创建GradeBook对象gradeBook1和gradeBook2，编译器必须知道一个GradeBook对象的大小。虽然概念上对象包含数据成员和成员函数，但是C++对象通常只包含数据。编译器仅创建类的成员函数的一份副本，该类所有的对象共享它。而每个对象毫无疑问地需要它自己的一份类的数据成员的副本，因为它们的内容随对象的不同而不同（例如，两个不同的BankAccount对象具有不同的数据成员balance）。可是，成员函数的代码是不可更改的。正因为如此，它们才可以被该类所有的对象所共享。因此，一个对象的大小依赖于存储类的数据成员所需的内存大小。通过在第7行包含GradeBook.h，我们使编译器获知确定一个GradeBook对象的大小所需要的信息（见图3.9中的第40行），以及确定类的对象是否被正确使用的信息（见图3.10中的第13~14行和第17~18行）。

第7行指示C++预处理器，在程序被编译前用GradeBook.h内容（即GradeBook的类定义）的副本替换这条指令。由于#include，当编译源代码文件fig03_10.cpp时，该文件已经包含了GradeBook的类定义，而且编译器能够确定如何创建GradeBook对象并查看它们的成员函数是否被正确调用。既然类的定义在头文件中（其中不包含main函数），那么我们可以在任何需要重用GradeBook类的程序中包含这个头文件。

如何找到头文件

请注意，图3.10第7行中的GradeBook.h头文件的名字是括在双引号（""）而非尖括号（<>）中的。正常情况下，程序的源代码文件和用户定义的头文件放在同一个目录下。当预处理器遇到括

分离接口与实现

在前面的例子中，每个类定义包含了类的 public 成员函数的完整定义及其私有数据成员的声明。可是，更好的软件工程是在类定义的外部定义成员函数，这样这些成员函数的实现细节对客户代码而言是隐藏的。这种方式保证程序员不会写出依赖于类的实现细节的客户代码。否则，若类的实现更改的话，客户代码将更可能“遭损坏”。

图 3.11 到图 3.13 的程序，通过将图 3.9 的类定义分成两个文件——定义 GradeBook 类的头文件 GradeBook.h（见图 3.11）和定义 GradeBook 成员函数的源代码文件 GradeBook.cpp（见图 3.12），分离了 GradeBook 类的接口和实现。按照惯例，成员函数的定义是放在一个与类的头文件基本名（例如，GradeBook）同名而文件扩展名是.cpp 的源代码文件中的。图 3.13 所示的源代码文件 fig03_13.cpp 定义了 main 函数（客户代码）。图 3.13 与图 3.10 的代码和输出是一样的。图 3.14 分别从 GradeBook 类的程序员和客户代码的程序员的角度，显示了由这 3 个文件构成的程序是如何编译的，我们将详细解释这张图。

GradeBook.h：用函数原型定义类的接口

图 3.11 所示的头文件 GradeBook.h 包含了 GradeBook 类定义的另一个版本（第 9~18 行）。该版本和图 3.9 的版本相似。但是，在图 3.9 中的函数定义在这里被函数原型所替换（第 12~15 行），它们描述类的公共接口而没有暴露类的成员函数的实现。函数原型是函数的声明，告诉编译器函数的名字、返回类型和形参的类型。请注意，头文件仍指定了类的 private 数据成员（第 17 行）。此外，编译器必须知道类的数据成员，以决定为类的每个对象保留多少内存。在客户代码中包含头文件 GradeBook.h（见图 3.13 中的第 8 行）给编译器提供了它需要的信息，保证客户代码正确调用 GradeBook 类的成员函数。

```
1  // Fig. 3.11: GradeBook.h
2  // GradeBook class definition. This file presents GradeBook's public
3  // interface without revealing the implementations of GradeBook's member
4  // functions, which are defined in GradeBook.cpp.
5  #include <string> // class GradeBook uses C++ standard string class
6  using std::string;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    GradeBook( string ); // constructor that initializes courseName
13    void setCourseName( string ); // function that sets the course name
14    string getCourseName(); // function that gets the course name
15    void displayMessage(); // function that displays a welcome message
16 private:
17    string courseName; // course name for this GradeBook
18 }; // end class GradeBook
```

图 3.11　包含指定类接口的函数原型的 GradeBook 的类定义

图 3.11 中第 12 行的函数原型指出这个构造函数需要一个 string 形参。记住构造函数是没有返回类型的，所以在这个函数原型中没有返回类型出现。成员函数 setCourseName 的函数原型（第 13 行）指出 setCourseName 需要一个 string 形参，并且不返回值（即它的返回类型是 void）。成员函数 getCourseName 的函数原型（第 14 行）指出该函数不需要形参，但返回一个字符串。

最后，成员函数 displayMessage 的函数原型（第 15 行）说明 displayMessage 不需要形参并且不返回任何值。除了不包含形参的名字（这在原型中是可选的），以及每个函数原型必须以分号结尾外，这些函数原型和图 3.9 中相应的函数头部一样。



常见的编程错误 3.8
在函数原型的末尾忘记加分号是一个语法错误。



良好的编程习惯 3.7
尽管函数原型中形参名是可选的（它们被编译器忽略），但是很多程序员出于编制文档的目的使用这些名字。



错误预防技巧 3.4
如果使用了错误的或容易混淆的名字，函数原型中的形参名（虽然编译器忽略它们）会令人误解。由于这个原因，许多程序员在能得到函数源代码的情况下，通过复制相应函数定义的第一行，然后在末尾添加分号的办法，创建函数原型。

GradeBook.cpp：在独立的源代码文件中定义成员函数

源代码文件GradeBook.cpp（见图3.12）定义了GradeBook类的成员函数，这些函数声明在图3.11中的第12~15行。成员函数的定义出现在第11~34行，与图3.9中第15~38行的成员函数定义几乎一样。

```
1  // Fig. 3.12: GradeBook.cpp
2  // GradeBook member-function definitions. This file contains
3  // implementations of the member functions prototyped in GradeBook.h.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "GradeBook.h" // include definition of class GradeBook
9
10 // constructor initializes courseName with string supplied as argument
11 GradeBook::GradeBook( string name )
12 {
13    setCourseName( name ); // call set function to initialize courseName
14 } // end GradeBook constructor
15
16 // function to set the course name
17 void GradeBook::setCourseName( string name )
18 {
19    courseName = name; // store the course name in the object
20 } // end function setCourseName
21
22 // function to get the course name
23 string GradeBook::getCourseName()
24 {
25    return courseName; // return object's courseName
26 } // end function getCourseName
27
28 // display a welcome message to the GradeBook user
29 void GradeBook::displayMessage()
30 {
31    // call getCourseName to get the courseName
32    cout << "Welcome to the grade book for\n" << getCourseName()
33       << "!" << endl;
34 } // end function displayMessage
```

图3.12　GradeBook成员函数的定义描述了GradeBook类的实现

请注意，在函数头部（第11、17、23和29行）中，每个成员函数名之前都添加了类名和符号“::”，该符号是二元作用域分辨运算符。通过这样的方式，将每个成员函数“捆绑”到目前分开的

GradeBook 的类定义上，该类定义声明了类的成员函数和数据成员。如果在每个函数名之前没有“GradeBook::”，编译器将不承认这些函数是GradeBook类的成员函数——编译器将认为它们是“自由的”或“无拘束的”函数，如main。这样的没有指定对象的函数不可以访问 GradeBook 的 private 数据，或者调用类的成员函数，所以编译器将不能编译这些函数。例如，访问变量 courseName 的第19和25行将导致编译错误，因为在每个函数中courseName没有被声明为局部变量，而且编译器不会知道 courseName 已经被声明为 GradeBook 类的数据成员。



常见的编程错误 3.9
当在类的外部定义类的成员函数时，在函数名前省略类名和二元作用域分辨运算符会导致编译错误。

为了指出 GradeBook.cpp 中的成员函数是 GradeBook 类的一部分，必须首先包含 GradeBook.h 头文件（见图3.12中的第8行）。这允许我们在GradeBook.cpp文件中访问类名GradeBook。当编译 GradeBook.cpp 时，编译器使用 GradeBook.h 中的信息保证：

1. 每个成员函数的第一行（第11、17、23和29行）匹配GradeBook.h中它的原型。例如，编译器保证 getCourseName 不接受形参，并返回一个字符串。
2. 每个成员函数了解类的数据成员和其他成员函数。例如，第19和25行可以访问变量 courseName，因为它是在 GradeBook.h 中声明的 GradeBook 类的一个数据成员，而第13行和第32行可以分别调用函数 setCourseName 和 getCourseName，因为它们都是在 GradeBook.h 中声明的类的成员函数，并且这些调用与相应的原型相符。

测试 GradeBook 类

图3.13执行同图3.10一样的GradeBook对象的处理。GradeBook的接口与其成员函数实现的分离不影响客户代码对类的使用方式，它只影响程序的编译和连接。我们很快将详细讨论后者。

```
1  // Fig. 3.13: fig03_13.cpp
2  // GradeBook class demonstration after separating
3  // its interface from its implementation.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "GradeBook.h" // include definition of class GradeBook
9
10 // function main begins program execution
11 int main()
12 {
13    // create two GradeBook objects
14    GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
15    GradeBook gradeBook2( "CS102 Data Structures in C++" );
16
17    // display initial value of courseName for each GradeBook
18    cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
19       << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
20       << endl;
21    return 0; // indicate successful termination
22 } // end main
```

```
gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++
```

图 3.13　接口和实现分离之后的 GradeBook 类的演示

与图3.10一样，图3.13中的第8行包含GradeBook.h文件，这样编译器可以保证在客户代码中正确地创建和操纵GradeBook对象。在执行这个程序之前，图3.12和图3.13中的源代码文件必须先经编译，然后连接在一起。就是说，客户代码中的成员函数的调用需要和类的成员函数的实现捆绑在一起。这是一项由连接器完成的工作。

编译和连接过程

图3.14是个示意图，显示了生成可供教师使用的、可执行的GradeBook应用程序的编译和连接过程。通常由一个程序员创建和编译类的接口和实现，而由不同的实现类的客户代码的程序员使用它们。因此，这张示意图显示了类实现程序员和客户代码程序员需要做的部分。图中虚线划分了类实现程序员、客户代码程序员和GradeBook应用程序用户各自需要做的部分。注意：图3.14不是一张UML图。

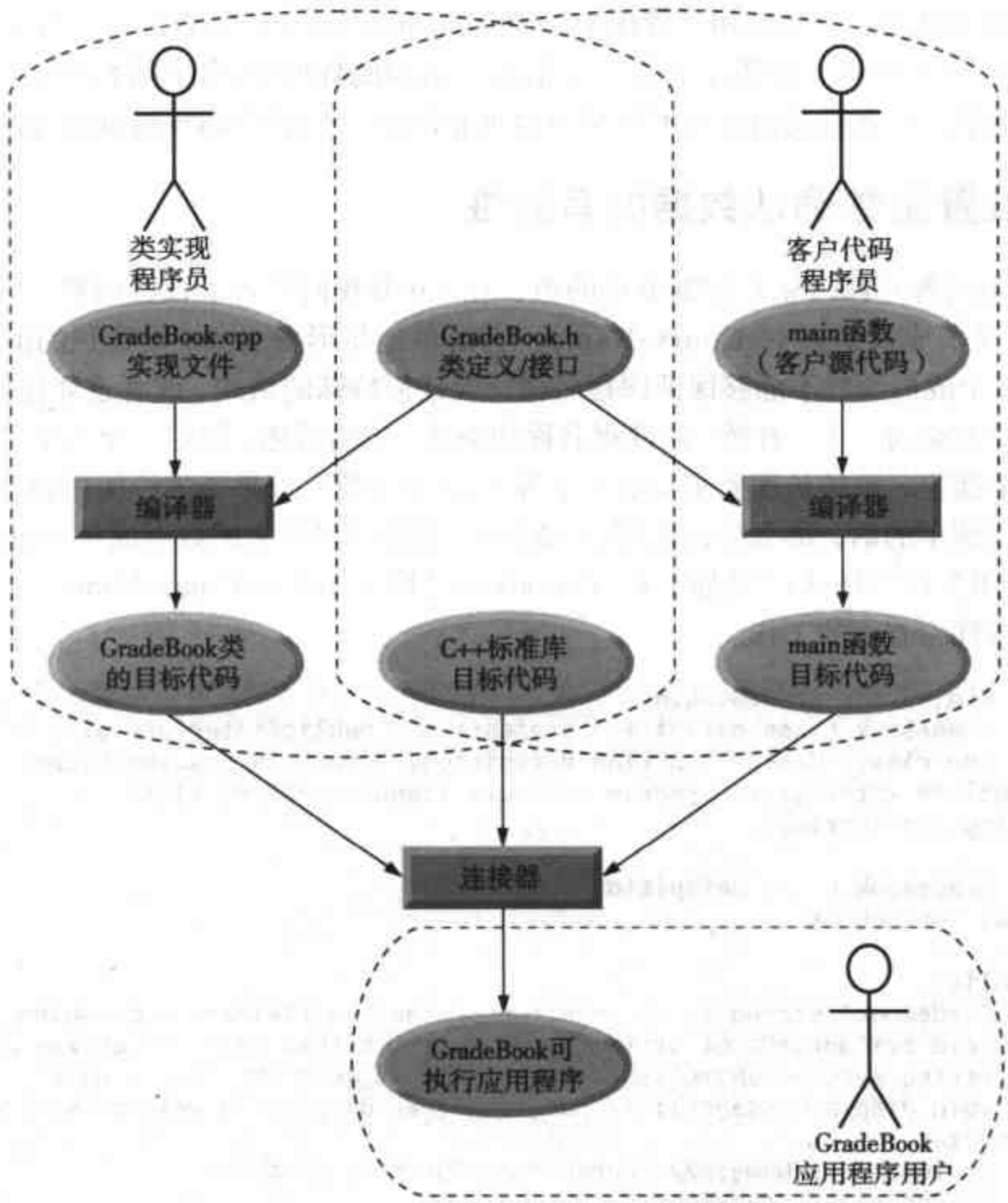


图 3.14　生成一个可执行程序的编译和连接过程

负责创建可重用GradeBook类的类实现程序员首先创建两个文件，一个是头文件GradeBook.h；另一个是#include（包含）该头文件的源代码文件GradeBook.cpp。然后，编译源代码文件创建GradeBook对象的目标代码。为了隐藏GradeBook成员函数的实现细节，类实现程序员将向客户代码程序员提供头文件GradeBook.h（它列出了类的接口和数据成员）和GradeBook类的目标代码，

目标代码包含了描述GradeBook成员函数的机器指令。GradeBook的源代码文件并没有交给客户代码程序员，所以客户依旧不知道 GradeBook 成员函数是如何实现的。

客户代码只需要知道使用GradeBook的接口，并保证能够连接到它的目标代码。由于类接口是GradeBook.h头文件中类定义的一部分，客户代码程序员必须有权访问这个文件，并在客户源代码中包含（#include）它。当编译客户代码时，编译器使用 GradeBook.h 中的类定义来保证 main 函数正确创建和操纵 GradeBook 类的对象。

为了创建供教师使用的 GradeBook 可执行应用程序，最后的步骤是要连接：

1. main 函数的目标代码（即客户代码）。
2. GradeBook 类成员函数实现的目标代码。
3. 类实现程序员和客户代码程序员使用的 C++ 类（例如，string）的 C++ 标准库目标代码。

连接器的输出就是可供教师用于管理学生成绩的 GradeBook 可执行的应用程序。

关于编译多个源代码文件程序的进一步信息，请参阅你所使用的编译器的文档，或者学习 Dive-Into 的出版物。后者是我们针对不同编译器所准备的，请访问 www.deitel.com/books/cpphtp5。

3.10 用设置函数确认数据的有效性

为了允许类的客户修改私有数据成员的值，在 3.6 节我们引入了设置函数。在图 3.5 中，类 GradeBook 定义了成员函数 setCourseName，它简单地将形参 name 接收到的值赋给数据成员 courseName。这个成员函数不能够保证课程名称符合任何特殊的格式，或者遵守任何其他的规则；这些格式和规则都是对一个“有效”的课程名称的描述。如前所述，假定一个大学可以打印学生成绩单，但是其中课程名称的长度要求只能小于等于 25 个字符。如果该学校使用包含 GradeBook 对象的系统生成成绩单的话，那么应该希望 GradeBook 类能够保证它的数据成员 courseName 决不超过 25 个字符。图 3.15 至图 3.17 增强了 GradeBook 类的成员函数 setCourseName，它执行有效性确认（也称为有效性检查）的工作。

```
1  // Fig. 3.15: GradeBook.h
2  // GradeBook class definition presents the public interface of
3  // the class. Member-function definitions appear in GradeBook.cpp.
4  #include <string> // program uses C++ standard string class
5  using std::string;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    GradeBook( string ); // constructor that initializes a GradeBook object
12    void setCourseName( string ); // function that sets the course name
13    string getCourseName(); // function that gets the course name
14    void displayMessage(); // function that displays a welcome message
15 private:
16    string courseName; // course name for this GradeBook
17 }; // end class GradeBook
```

图 3.15 GradeBook 的类定义

GradeBook 的类定义

请注意，图 3.15 中的 GradeBook 的类定义，特别是它的接口，都同图 3.11 中的一模一样。由于接口保持不变，当成员函数 setCourseName 的定义被修改时，这个类的客户也不必改变。这样，客户就可以简单地通过将客户代码连接到更新后的GradeBook目标代码来利用改进后的GradeBook类。

用 GradeBook 成员函数 setCourseName 确认课程名称的有效性

GradeBook 类的增强体现在 setCourseName 的定义中（见图 3.16 中的第 18~31 行）。第 20~21 行中的 if 语句确定形参 name 是否包含一个有效的课程名称（即一个至多含 25 个字符的字符串）。如果课程名称是有效的，那么第 21 行将它存放到数据成员 courseName 中。请注意，第 20 行中的表达式 name.length()，它和 myGradeBook.displayMessage()一样是一个成员函数调用。C++ 标准库的 string 类定义了成员函数 length，它返回一个 string 对象中的字符个数。形参 name 是一个 string 对象，所以调用 name.length()返回的是 name 中的字符个数。如果返回值小于或等于 25，那么 name 是有效的，于是执行第 21 行。

```
1  // Fig. 3.16: GradeBook.cpp
2  // Implementations of the GradeBook member-function definitions.
3  // The setCourseName function performs validation.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "GradeBook.h" // include definition of class GradeBook
9
10 // constructor initializes courseName with string supplied as argument
11 GradeBook::GradeBook( string name )
12 {
13    setCourseName( name ); // validate and store courseName
14 } // end GradeBook constructor
15
16 // function that sets the course name;
17 // ensures that the course name has at most 25 characters
18 void GradeBook::setCourseName( string name )
19 {
20    if ( name.length() <= 25 ) // if name has 25 or fewer characters
21       courseName = name; // store the course name in the object
22
23    if ( name.length() > 25 ) // if name has more than 25 characters
24    {
25       // set courseName to first 25 characters of parameter name
26       courseName = name.substr( 0, 25 ); // start at 0, length of 25
27
28       cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
29          << "Limiting courseName to first 25 characters.\n" << endl;
30    } // end if
31 } // end function setCourseName
32
33 // function to get the course name
34 string GradeBook::getCourseName()
35 {
36    return courseName; // return object's courseName
37 } // end function getCourseName
38
39 // display a welcome message to the GradeBook user
40 void GradeBook::displayMessage()
41 {
42    // call getCourseName to get the courseName
43    cout << "Welcome to the grade book for\n" << getCourseName()
44       << "!" << endl;
45 } // end function displayMessage
```

图 3.16　GradeBook类的成员函数定义，其中使用了确认数据成员 courseName 长度有效性的一个设置函数

```
1  // Fig. 3.17: fig03_17.cpp
2  // Create and manipulate a GradeBook object; illustrate validation.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "GradeBook.h" // include definition of class GradeBook
8
9  // function main begins program execution
10 int main()
11 {
12    // create two GradeBook objects;
13    // initial course name of gradeBook1 is too long
14    GradeBook gradeBook1( "CS101 Introduction to Programming in C++" );
15    GradeBook gradeBook2( "CS102 C++ Data Structures" );
16
17    // display each GradeBook's courseName
18    cout << "gradeBook1's initial course name is: "
19       << gradeBook1.getCourseName()
20       << "\ngradeBook2's initial course name is: "
21       << gradeBook2.getCourseName() << endl;
22
23    // modify myGradeBook's courseName (with a valid-length string)
24    gradeBook1.setCourseName( "CS101 C++ Programming" );
25
26    // display each GradeBook's courseName
27    cout << "\ngradeBook1's course name is: "
28       << gradeBook1.getCourseName()
29       << "\ngradeBook2's course name is: "
30       << gradeBook2.getCourseName() << endl;
31    return 0; // indicate successful termination
32 } // end main
```

```
Name "CS101 Introduction to Programming in C++" exceeds maximum length (25).
Limiting courseName to first 25 characters.

gradeBook1's initial course name is: CS101 Introduction to Pro
gradeBook2's initial course name is: CS102 C++ Data Structures

gradeBook1's course name is: CS101 C++ Programming
gradeBook2's course name is: CS102 C++ Data Structures
```

图3.17　创建和操作一个GradeBook对象，该对象中的课程名称限制到25个字符长

图3.17中的第18~21行显示gradeBook1的已截短的课程名称和gradeBook2的课程名称。第24行直接调用gradeBook1的setCourseName成员函数，将这个GradeBook对象的课程名称更改为更短的不需截短的名字。接着，第27~30行再次输出每个GradeBook对象的课程名称。

关于设置函数的补充说明

像setCourseName之类的public的设置函数，应当仔细审查对数据成员（例如，courseName）值的任何修改尝试，以保证新的值适合该数据项。例如，应当拒绝将月的天数设置为37，将人的体重设置为0或一个负值，或者将考试成绩设置为185（当正确的范围是0到100时）等的各种企图。



软件工程知识3.6

把数据成员设置成private的，而由public的成员函数控制访问数据成员的权利，尤其是写的权利，将有助于保证数据的完整性。

伪代码语句通常只描述可执行语句，所谓可执行语句是指程序员将程序从伪代码转换成C++代码并在计算机上运行时，会引起特定动作发生的语句。没有涉及到初始化和构造函数调用的声明并不是可执行语句。例如，下面的声明

```
int i;
```

只告诉编译器变量i的类型，并指示编译器在内存中为这个变量预留内存空间。但是，当程序执行时，这个声明并不会导致任何动作，比如输入、输出或一次计算等的发生。通常，在我们的伪代码中不包括变量声明。然而，有些程序员也会在伪代码程序的开始部分，列出变量并提到它们的用途。

现在，我们看一个伪代码的例子，该例是为帮助程序员创建图2.5的相加程序而书写的。如图4.1所示，此伪代码对应这样的算法：首先由用户输入两个整数，然后将这两数相加并显示其和。虽然我们在此展示了完整的伪代码，可是在本章稍后部分，我们还会演示如何从问题陈述出发创建伪代码。

```
1  Prompt the user to enter the first integer
2  Input the first integer
3
4  Prompt the user to enter the second integer
5  Input the second integer
6
7  Add first integer and second integer, store result
8  Display result
```

图4.1　图2.5相加程序的伪代码

图4.1的第1~2行对应图2.5的第13~14行语句。请注意，伪代码语句是简单的英语语句，表达在C++中所执行的任务。同样，第4~5行对应图2.5的第16~17行语句，第7~8行对应图2.5中的第19和21行语句。

图4.1中的伪代码有几个值得关注的重要方面。注意，伪代码只对应函数main的代码，这是因为伪代码通常针对算法，而非完整的程序。既然这样，伪代码用于表示算法。代码所在的函数对算法本身来讲并不重要。同理，在伪代码中不包括图2.5的第23行（返回语句）——这条返回语句放置在每个main函数的结束部分，对算法并不重要。最后，在伪代码中也不包括图2.5中的第9~11行，因为这些变量声明不是可执行语句。

4.4　控制结构

一般说来，程序中的语句是按照书写的顺序逐条执行的。这种程序执行的方式称为顺序执行。可是，很快要讨论的许多C++语句还允许程序员自己指定接下来要执行的语句，该语句也许不是顺序的下一条语句。这种程序执行的方式称为控制转移。

在20世纪60年代，事实已证明软件开发小组所经历的很多挫折，其根源在于对控制转移不加选择的使用。指责的矛头直指goto语句。利用这条语句，程序员几乎可以将控制权转交给程序中任意地方（从而创建了常被称为“意大利面条式的代码”）。因此，所谓的结构化程序的概念几乎成了“取消goto”的同义词。

Böhm和Jacopini的研究证明，编写程序完全可以不用任何goto语句。大势所趋之下，程序员们逐渐放弃对goto的偏爱，逐渐转向“无goto编程”。直到20世纪70年代，程序员们才开始认真

看待结构化编程。结果令人振奋，软件开发小组的报告显示，采用结构化编程后，开发时间降低了，系统的按时递交率和软件项目的按预算完成率提高了。这些成功的关键是结构化程序条理更清晰，更易调试、测试和修改，而且更有可能在第一时间避免错误。

Böhm 和 Jacopini 的工作还证明，编写所有的程序其实只需 3 种控制结构，即顺序结构、选择结构和循环结构。术语“控制结构”来自于计算机科学领域。当我们介绍控制结构的 C++ 实现时，将按照 C++ 标准文档的术语称它们为“控制语句”。

C++ 中的顺序结构

顺序结构是 C++ 内置的。除非特别声明，计算机总是按照 C++ 语句书写的顺序逐条执行它们——换句话说，就是顺次执行。图 4.2 的统一建模语言（UML）活动图举例说明的便是一个典型的顺序结构，两个计算将依次执行。很快我们将会看到，在任何可以放置单一动作的地方，都可以放置顺次的若干动作。

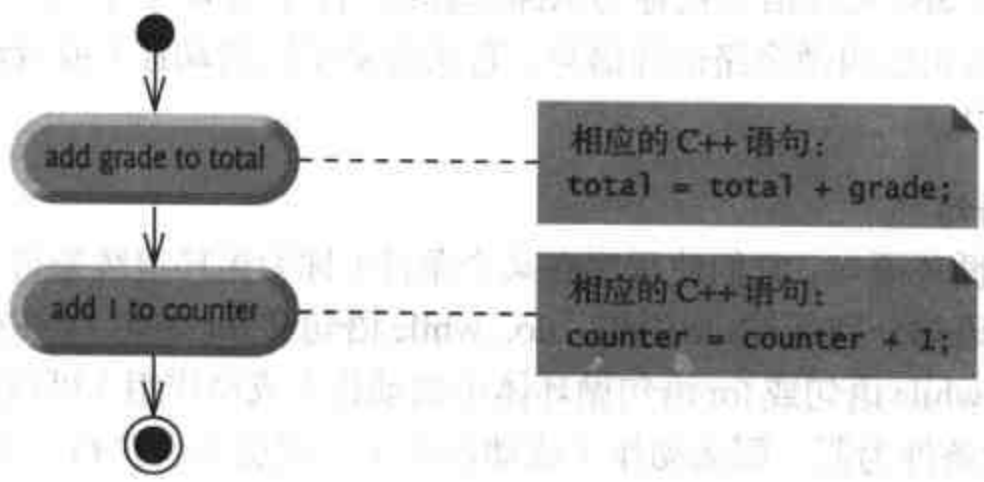


图 4.2　顺序结构活动图

在这个图中，两条语句分别是将一个成绩（grade）加到累加和变量 total 中，以及对计数器变量 counter 加 1。这样的语句可能出现在求若干学生成绩的平均值的程序中。为了计算平均值，所要平均的成绩之和要除以成绩的个数。计数器变量用来记录参与平均的值的个数。在 4.8 节的程序中将看到类似的语句。

活动图是 UML 的一部分。一张活动图是对部分软件系统的工作流，也称为活动（activity）的建模。这样的工作流可能包括算法的一部分，例如图 4.2 中的顺序结构。活动图由一些具有特殊意义的符号组成，例如动作状态符号（圆角矩形）、菱形和小圆圈等。这些符号通过转换箭头相连接，表示活动的流向。

虽然许多程序员更喜欢伪代码，但和伪代码一样，活动图可以帮助程序员开发和表示算法。它能够清楚地显示出控制结构的运作机制。

考虑图 4.2 的顺序结构活动图。它包含两个动作状态，表示要执行的动作。每个动作状态都包含一个动作表达式，例如，“加 grade 到 total 中”或“counter 加 1”都指定了要执行的特定动作。其他的动作可能包括计算或输入/输出等操作。在活动图中的箭头称为转换箭头。这些箭头表示转换，指出了由动作状态符号表示的动作所出现的顺序——实现图 4.2 活动图描述的活动的程序，首先将 grade 累加到 total，然后让 counter 加 1。

位于活动图顶部的实心圆圈表示了活动的初始状态——程序执行已建模的活动之前工作流的开始位置，而位于活动图底部的一个空心圆圈围绕的实心圆圈表示了结束状态——程序执行其活动之后工作流的结束位置。

请注意，C++代码与伪代码是严格对应的。正是由于伪代码的这一性质，才使它成为一种强有力的开发工具。

图4.4示意了单路选择的if语句。它包含了可能是活动图中最重要的符号——菱形符号，或者称判定符号，指出需要做出一个判断。判定符号表明，工作流将沿着一条决定了的路径前进。该路径由与判定符号相联系的监控条件所决定，这个条件可以为真，也可以为假。从判定符号出发的每个转换箭头都有一个监控条件（附着于转换箭头上或者旁边）。如果某个特定的监控条件为真，工作流则进入转换箭头所指的动作状态。在图4.4中，如果成绩大于或等于60分，那么程序在屏幕上打印 “Passed”（“通过”），然后转向该活动的结束状态；如果成绩小于60分，那么程序立即转向结束状态，并不显示任何信息。

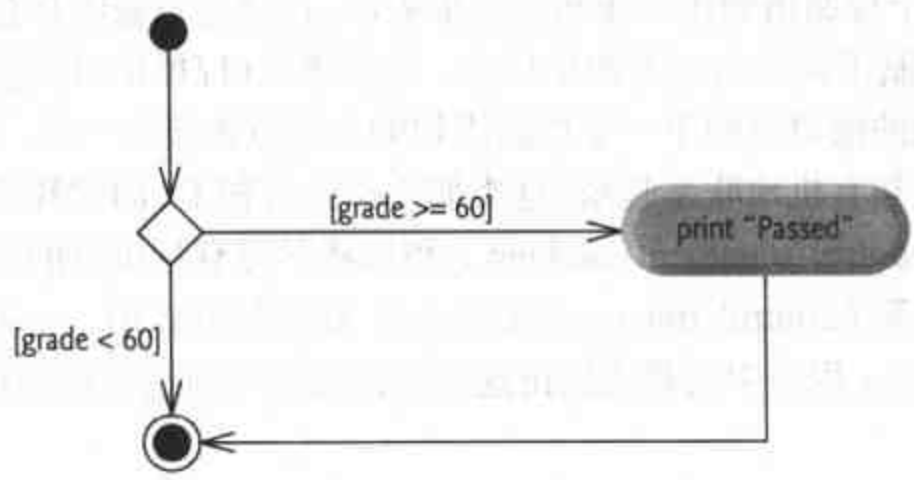


图 4.4　if单路选择语句的活动图

在第1章中我们曾介绍过，可以根据包含了关系或相等运算符的条件做出判断。实际上，在C++中，可以基于任何表达式做出判断——如果表达式的值等于零，就把它当做假；如果表达式的值不等于零，就把它当做真。C++提供了bool（布尔）数据类型，这种数据类型的变量其取值只能是true（意为真）或false（意为假）——true和false是C++的两个关键字。



可移植性提示 4.1

为了与用整数表示布尔值的早期C版本兼容，bool值true也可以用任何非零值表示（编译器通常使用1），而bool值false也可以用零值表示。

请注意，if语句是一条单入口/单出口语句。随着学习的深入，我们将看到对于其他的控制语句，其活动图同样包含初始状态、转换箭头、用以表明执行动作的动作状态、用于做出判断的判定符号（与监控条件相关）和结束状态等。这与我们已经在强调的动作/判断程序设计模型是一致的。

我们可以想像有7个柜子，每个柜子包含大量相同类型的空UML活动图，这里的类型是指前述的7种形式的控制语句，即7个柜子分别对应7种形式的控制语句。于是程序员的任务就是用惟一可选的两种方法（堆叠或者嵌套），根据算法需要，从柜子中挑选出最恰当的、尽可能多的每种控制语句的活动图来装配程序，然后以满足算法结构化实现的方式，在动作状态和判定中填入动作表达式和监控条件。稍后我们将讨论各种用于书写动作和判定的方式。

4.6　if...else 双路选择语句

只有当条件为true时，if单路选择语句才会执行指定的动作；否则，便跳过该动作。利用if...else双路选择语句，程序员可以根据条件的真假，指定不同的执行动作。也就是说，当条件为true时，执行一个动作；条件为false时，执行另一个不同的动作。例如，下面的伪代码语句：

```
If student's grade is greater than or equal to 60（如果学生的成绩大于或等于 60 分）
    Print "Passed"（打印 "通过"）
Else（否则）
    Print "Failed"（打印 "未通过"）
```

在学生的成绩大于或等于60分时，打印"Passed"，而当学生的成绩小于60分时，打印"Failed"。无论哪种情况发生，在完成打印操作后，都会依次执行下一条伪代码语句。

上述 If...Else 伪代码语句若用 C++ 代码书写的话，则为如下正式的 if...else 语句：

```
if ( grade >= 60 )
   cout << "Passed";
else
   cout << "Failed";
```

请注意，else 的体也进行了缩进处理。不管采用什么样的缩进约定，都应当在整个程序中保持约定的一致性。如果不遵守统一的缩进约定，程序的可读性会很差。



良好的编程习惯 4.2
if/else 语句的两个体语句都应该采用缩进格式。



良好的编程习惯 4.3
如果有多级缩进，那么每级相对上级缩进的空格幅度应保持一致。

图4.5描述了if...else语句的控制流程。再一次请注意，除了初始状态、转换箭头和结束状态以外，活动图中用了只表示动作状态和判断的其他符号。为了加深印象，我们会继续强调这种动作/判断计算模型。再次想像在一个大柜子中包含了大量程序员使用的、空的双路语句的UML活动图，程序员的任务便是采用堆叠和嵌套的方式，将这些双路语句活动图同其他控制语句的活动图组合起来，以形成算法的结构化实现。然后，程序员再用适合于算法的动作表达式和监控条件填写动作状态和判定符号。

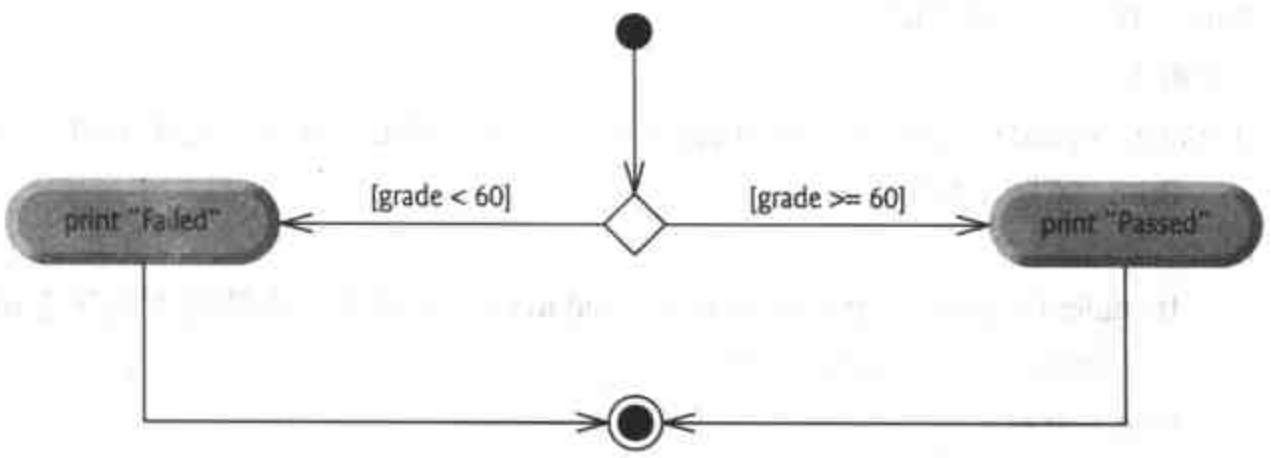


图 4.5　if...else 双路选择语句的活动图

条件运算符（?:）

C++ 提供了条件运算符(?:)，它与 if...else 语句有着非常密切的关系。条件运算符是 C++ 惟一的三元运算符——它需要三个操作数。这些操作数和条件运算符一起构成了条件表达式。其中，第一个操作数是一个条件，第二个操作数是条件为true时整个条件表达式的值，第三个操作数是条件为 false 时整个条件表达式的值。例如，下面的输出语句：

```
cout << ( grade >= 60 ? "Passed" : "Failed" );
```

中。判定符号和合并符号可以通过检查“流入”和“流出”的转换箭头的个数来区分。判定符号只有一个转换箭头指向菱形符号，但有两个或者更多个转换箭头从菱形中流出来，用以指明可能的流向。另外，从判定符号出发的每个转换箭头上都附有一个监控条件。合并符号则是有两个或者更多个转换箭头流入菱形符号，而只有一个转换箭头从菱形符号中流出来，用以表明接下来的活动是之前多个活动工作流的融合。请注意，与判定符号不同的是，合并符号并没有相应的C++对应物。与合并符号相关的转换箭头上也没有附带的监控条件。

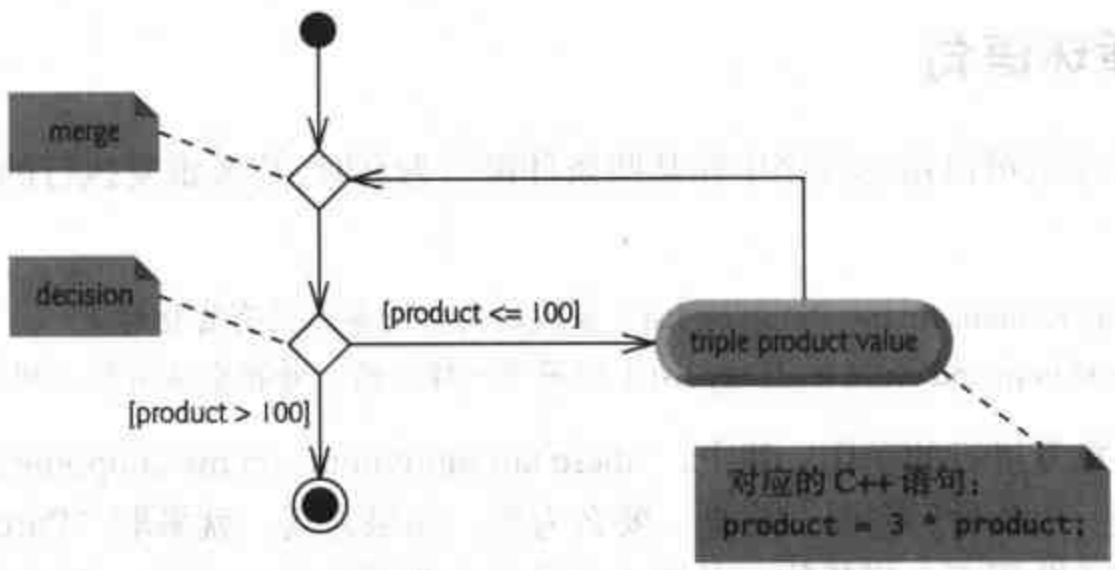


图 4.6　while 循环语句的 UML 活动图

图 4.6 的活动图非常清晰地揭示了本节前面讨论的 while 语句的循环机理。从动作状态出发的转换箭头指向合并符号，而合并符号的转换箭头会重新返回判定符号。这个判定符号所表示的判断在每次循环中被检测，直到监控条件 product > 100 变为真为止。于是，该 while 语句退出（即，到达其结束状态），并将控制权移交给程序中顺次的下一条语句。

想像一下，在一个大柜子中包含了大量程序员使用的、空的while循环语句的UML活动图——程序员的任务便是采用堆叠和嵌套的方式，将这些while循环语句同其他控制语句的活动图组合起来，以形成算法的结构化实现。然后，程序员再用适合于算法的动作表达式和监控条件填写动作状态和判定符号。



性能提示 4.3

在本书中，我们提及的许多性能提示都只能稍微改进程序的性能，因此，读者可能会倾向于忽略它们。但是，对于在循环中多次执行的代码而言，对它们的微小的性能改进可能导致实质性的整体性能的改进。

4.8　算法详述：计数器控制的循环

为了说明程序员是如何开发算法的，本节和4.9节分别以求全班平均成绩问题的两种具体形式为例，阐述相应的两种解决办法。本节考虑的问题陈述是：

> 一个有 10 名学生的班级进行了一次测验，测验的成绩（0~100 中的一个整数）现在都已经打印出了。请计算并打印该班学生的总成绩以及班级的平均成绩。

我们知道，班级的平均成绩等于总成绩除以学生的人数。从上面的问题陈述，我们可以分析出：在计算机上解决该问题的算法必须包括输入每个学生的成绩，计算平均成绩以及打印结果三大步骤。

计数器控制的循环的伪代码算法

下面用伪代码列出要执行的动作，并指定这些动作执行的顺序。我们使用计数器控制的循环以一次一个的方式输入每个学生的成绩。这种方法用一个称为计数器（counter）的变量控制一组语句执行的次数（也称为循环的迭代次数）。

我们常常将计数器控制的循环称为定数循环（definite repetition），因为在循环开始执行之前，循环的次数是已知的。在本例中，当这种计数器超过 10 时，循环结束。这一节给出了一个已完全开发好的伪代码算法（如图 4.7 所示），以及一个版本的 GradeBook 类（如图 4.8 和图 4.9 所示）。其中，GradeBook 类在一个 C++ 成员函数中实现了该算法。之后，本节还给出了一个演示该算法执行的应用程序（如图 4.10 所示）。在 4.9 节中，我们将示范如何用伪代码从头开始开发出这样的一个算法。



软件工程知识 4.3

经验表明，在计算机上解决问题最困难的部分在于开发出解决该问题的算法。一旦确定了正确算法，从算法生成 C++ 程序通常是相当简单的。

请注意，图 4.7 的伪代码算法中涉及到的总数和计数器。总数（total）是一个变量，用于累计一系列数值的和，计数器（counter）也是一个变量，用来计数。在这个例子中，成绩计数器用来指示用户正打算输入第几个成绩（一共 10 个成绩）。用来存放总数的变量在程序中使用之前，一般被初始为 0；否则，和将包含总数内存区域中存放的以前的数值。

```
1  Set total to zero
2  Set grade counter to one
3
4  While grade counter is less than or equal to ten
5      Prompt the user to enter the next grade
6      Input the next grade
7      Add the grade into the total
8      Add one to the grade counter
9
10 Set the class average to the total divided by ten
11 Print the total of the grades for all students in the class
12 Print the class average
```

图 4.7　采用计数器控制的循环解决求全班平均成绩问题的伪代码算法

```
1  // Fig. 4.8: GradeBook.h
2  // Definition of class GradeBook that determines a class average.
3  // Member functions are defined in GradeBook.cpp
4  #include <string> // program uses C++ standard string class
5  using std::string;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    GradeBook( string ); // constructor initializes course name
12    void setCourseName( string ); // function to set the course name
13    string getCourseName(); // function to retrieve the course name
14    void displayMessage(); // display a welcome message
15    void determineClassAverage(); // averages grades entered by the user
16 private:
17    string courseName; // course name for this GradeBook
18 }; // end class GradeBook
```

图 4.8　采用计数器控制的循环解决求全班平均成绩的问题：GradeBook 的头文件

```
1  // Fig. 4.9: GradeBook.cpp
2  // Member-function definitions for class GradeBook that solves the
3  // class average program with counter-controlled repetition.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include "GradeBook.h" // include definition of class GradeBook
10
11 // constructor initializes courseName with string supplied as argument
12 GradeBook::GradeBook( string name )
13 {
14    setCourseName( name ); // validate and store courseName
15 } // end GradeBook constructor
16
17 // function to set the course name;
18 // ensures that the course name has at most 25 characters
19 void GradeBook::setCourseName( string name )
20 {
21    if ( name.length() <= 25 ) // if name has 25 or fewer characters
22       courseName = name; // store the course name in the object
23    else // if name is longer than 25 characters
24    { // set courseName to first 25 characters of parameter name
25       courseName = name.substr( 0, 25 ); // select first 25 characters
26       cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
27          << "Limiting courseName to first 25 characters.\n" << endl;
28    } // end if...else
29 } // end function setCourseName
30
31 // function to retrieve the course name
32 string GradeBook::getCourseName()
33 {
34    return courseName;
35 } // end function getCourseName
36
37 // display a welcome message to the GradeBook user
38 void GradeBook::displayMessage()
39 {
40    cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
41       << endl;
42 } // end function displayMessage
43
44 // determine class average based on 10 grades entered by user
45 void GradeBook::determineClassAverage()
46 {
47    int total; // sum of grades entered by user
48    int gradeCounter; // number of the grade to be entered next
49    int grade; // grade value entered by user
50    int average; // average of grades
51
52    // initialization phase
53    total = 0; // initialize total
54    gradeCounter = 1; // initialize loop counter
55
56    // processing phase
57    while ( gradeCounter <= 10 ) // loop 10 times
58    {
59       cout << "Enter grade: "; // prompt for input
60       cin >> grade; // input next grade
61       total = total + grade; // add grade to total
```

图 4.9　采用计数器控制的循环解决求全班平均成绩的问题：GradeBook 的源代码文件

```
62          gradeCounter = gradeCounter + 1; // increment counter by 1
63       } // end while
64
65       // termination phase
66       average = total / 10; // integer division yields integer result
67
68       // display total and average of grades
69       cout << "\nTotal of all 10 grades is " << total << endl;
70       cout << "Class average is " << average << endl;
71    } // end function determineClassAverage
```

图 4.9　采用计数器控制的循环解决求全班平均成绩的问题：GradeBook 的源代码文件（续）

增强 GradeBook 的有效性确认

在讨论全班平均成绩算法的实现之前，首先仔细考虑一下我们对GradeBook类的增强工作。在图3.16中，成员函数setCourseName将验证课程名称的合法性。它首先用一条if语句测试课程名称的长度是否小于或者等于25个字符。如果是这样，接受该课程名称。紧接着这条if语句的是另一条if语句，它测试课程名称的长度是否大于25个字符（在这种情况下，该课程名称将被缩短）。请注意，第二条if语句的条件与第一条if语句的条件是恰恰相反的。如果一条if语句的条件的值是true的话，那么另一条的必定为false。这正是采用一条if...else语句的理想状况，因而我们在这里改进了代码，将图3.16中的这两条if语句用一条if...else语句代替（见图4.9中的第21~28行）。

在 GradeBook 类中实现计数器控制的循环

GradeBook类（如图4.8和图4.9所示）包含了一个构造函数。它声明在图4.8的第11行，而定义在图4.9的第12~15行，作用是对声明在图4.8中第17行的该类的实例变量courseName赋值。图4.9的第19~29行、第32~35行和第38~42行分别定义了该类的成员函数setCourseName、getCourseName和displayMessage；第45~71行定义了成员函数determineClassAverage，这个函数实现了图4.7中用伪代码描述的求全班平均成绩的算法。

第47~50行声明了int类型的局部变量total、gradeCounter、grade和average，其中，变量grade用于存放用户输入的一次成绩。请注意，上述的这些变量都是声明在成员函数determineClassAverage的函数体内的。

在本章的GradeBook类的版本中，我们简单地读入和处理了一组学生成绩。对平均分的计算只在成员函数determineClassAverage中进行，即只用到了局部变量，并没有在类的实例变量（即，数据成员）中保存关于所有学生成绩的任何信息。在第7章中，我们将对GradeBook类继续进行改进，将输入的所有成绩用一个实例变量保存起来，此变量是一个称为数组的数据结构。这样，GradeBook类的对象就可以对这组成绩进行各种各样的计算，而不需要用户一遍遍地输入该组数据。



良好的编程习惯 4.5
将声明语句与其他语句用空行隔开可以增强可读性。

第53~54行初始化total为0，gradeCounter为1。请注意，变量total和gradeCounter在用于计算前已经被初始化了。计数器变量一般初始化为0或者1，取决于它们的用法（对此我们将举例说明）。未初始化的变量中存放的是“垃圾”值（“garbage” value），也称为不确定的值，即为此变量分配的内存区域中原先存储的值。变量grade和average（分别为用户输入和计算出的平均值而准备）在这里不需要初始化，它们将分别在函数输入数据和计算出结果时被赋值。



常见的编程错误 4.6
没有初始化的各种计数器变量和总数变量将导致逻辑错误。



错误预防技巧 4.2

初始化每一个计数器变量和总数变量，或者在声明时，或者在赋值语句中。总数变量通常被初始化为0。根据用法的不同，计数变量一般被初始化为0或1（后面会有何时用0和何时用1的例子）。



良好的编程习惯 4.6

每一行声明一个变量并且加上注释，这可以增强程序可读性。

第57行表明只要gradeCounter的值小于或者等于10，while语句将继续循环（也称为迭代）。换句话说，当这一条件仍为真时，这条while语句会重复地执行用花括号括起来的循环体（见第58~63行）中的各条语句。

第59行显示提示"Enter grade:",实际对应了伪代码语句“Prompt the user to enter the next grade”。第60行读入用户输入的成绩并把它赋给变量grade。当然，这一行对应了伪代码语句 “Input the next grade”。前面曾提到过，在程序中变量 grade 一开始并没有被初始化，因为程序在每一次循环迭代期间从用户那里为 grade 获取值。第 61 行将用户输入的新成绩与 total 相加，并将结果再赋给 total，这样意味着 total 原来的值被新的结果值替换了。

第62行将计数器变量gradeCounter增1，表示程序已经处理了一个成绩，并且准备接受用户输入的下一个成绩。gradeCounter每次的增1，最终造成该变量的值超过10。这时候，while循环终止，因为其条件（第 57 行）变成了假。

当循环终止时，第66行执行平均值的计算，并将结果赋给变量average。第69行显示文本 “Total of all 10 grades is”（所有10个成绩的总和是），之后是变量total的值。第70行显示文本 “Class average is ”（全班的平均成绩是），紧接着是变量 average 的值。然后，成员函数 determineClassAverage 将控制权还给主调函数（即，图 4.10 中的 main 函数）。

```
1  // Fig. 4.10: fig04_10.cpp
2  // Create GradeBook object and invoke its determineClassAverage function.
3  #include "GradeBook.h" // include definition of class GradeBook
4
5  int main()
6  {
7     // create GradeBook object myGradeBook and
8     // pass course name to constructor
9     GradeBook myGradeBook( "CS101 C++ Programming" );
10
11    myGradeBook.displayMessage(); // display welcome message
12    myGradeBook.determineClassAverage(); // find average of 10 grades
13    return 0; // indicate successful termination
14 } // end main
```

```
Welcome to the grade book for
CS101 C++ Programming

Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

图 4.10　用计数器控制的循环解决求全班平均成绩的问题：创建 GradeBook 类（见图4.8和图4.9）的一个对象并调用其成员函数determineClassAverage

```
Initialize total to zero
Initialize counter to zero

Prompt the user to enter the first grade
Input the first grade (possibly the sentinel)

While the user has not yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Prompt the user to enter the next grade
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the total of the grades for all students in the class
    Print the class average
else
    Print "No grades were entered"
```

图 4.11　采用标记控制的循环的求全班平均成绩问题的伪代码算法

在图 4.11 中的伪代码算法可以解决更一般化的求全班平均成绩的问题。这个算法的产生只经历了两级细化过程。有时，可能需要更多级的细化才能得到理想的算法。



软件工程知识 4.6

当伪代码算法描述的细节足以将伪代码转换成 C++ 代码时，这种自顶向下逐步求精的过程便可结束了。在正常情况下，这时 C++ 程序的实现是件水到渠成的事。



软件工程知识 4.7

很多经验丰富的程序员编写程序时根本不使用诸如伪代码之类的程序开发工具。这些程序员认为其最终目标是解决计算机上的问题，编写伪代码只会推迟产品的推出。尽管这种开发方式在简单的或者熟悉的问题上能行得通，但是在大型的或者复杂的项目中往往会导致开发陷入困境。

在 GradeBook 类中实现标记控制的循环

图 4.12 和图 4.13 给出了包含成员函数 determineClassAverage 的 GradeBook 类的 C++ 源代码，该成员函数实现了图 4.11 中的伪代码算法（图 4.14 是这个类使用的示范）。尽管每次输入的成绩都是整数，但是它们的平均值很有可能是一个带小数点的数——也就是说，一个实数或者叫做浮点数，例如 7.33,0.0975 或者 1 000.12345，等等。数据类型 int 不能表示这样的数，因此，这个类必须采用另外一种数据类型来做这项工作。C++ 提供多种数据类型来在内存中存放浮点数，包括 float（单精度浮点）类型和 double（双精度浮点）类型。这两种类型之间的主要区别在于：与 float 类型的变量比较起来，double 类型的变量能够存放具有更大数量级和更高精度（也就是说，小数点后可以有更多位数字，即数的精度）的数。这里的程序引入了一种特殊的运算符——强制类型转换运算符，迫使求平均数的计算产生一个浮点数的结果。这些特征的具体细节将在讨论程序时说明。

在这个例子中我们看到，就像小孩堆积木一样，一条控制语句可以依次堆叠到另一条控制语句上。在图 4.13 中，第 67~75 行的 while 语句之后紧随的是一条 if...else 语句。由于该程序的大部分代码与图 4.9 中的代码相同，所以我们侧重解释其新特征和新内容。

```
1  // Fig. 4.12: GradeBook.h
2  // Definition of class GradeBook that determines a class average.
3  // Member functions are defined in GradeBook.cpp
4  #include <string> // program uses C++ standard string class
5  using std::string;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    GradeBook( string ); // constructor initializes course name
12    void setCourseName( string ); // function to set the course name
13    string getCourseName(); // function to retrieve the course name
14    void displayMessage(); // display a welcome message
15    void determineClassAverage(); // averages grades entered by the user
16 private:
17    string courseName; // course name for this GradeBook
18 }; // end class GradeBook
```

图4.12　采用标记控制的循环解决求全班平均成绩的问题：GradeBook的头文件

```
1  // Fig. 4.13: GradeBook.cpp
2  // Member-function definitions for class GradeBook that solves the
3  // class average program with sentinel-controlled repetition.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8  using std::fixed; // ensures that decimal point is displayed
9
10 #include <iomanip> // parameterized stream manipulators
11 using std::setprecision; // sets numeric output precision
12
13 // include definition of class GradeBook from GradeBook.h
14 #include "GradeBook.h"
15
16 // constructor initializes courseName with string supplied as argument
17 GradeBook::GradeBook( string name )
18 {
19    setCourseName( name ); // validate and store courseName
20 } // end GradeBook constructor
21
22 // function to set the course name;
23 // ensures that the course name has at most 25 characters
24 void GradeBook::setCourseName( string name )
25 {
26    if ( name.length() <= 25 ) // if name has 25 or fewer characters
27       courseName = name; // store the course name in the object
28    else // if name is longer than 25 characters
29    { // set courseName to first 25 characters of parameter name
30       courseName = name.substr( 0, 25 ); // select first 25 characters
31       cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
32          << "Limiting courseName to first 25 characters.\n" << endl;
33    } // end if...else
34 } // end function setCourseName
35
36 // function to retrieve the course name
37 string GradeBook::getCourseName()
38 {
39    return courseName;
40 } // end function getCourseName
41
```

图4.13　采用标记控制的循环解决求全班平均成绩的问题：GradeBook的源代码文件

```
42 // display a welcome message to the GradeBook user
43 void GradeBook::displayMessage()
44 {
45    cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
46       << endl;
47 } // end function displayMessage
48
49 // determine class average based on 10 grades entered by user
50 void GradeBook::determineClassAverage()
51 {
52    int total; // sum of grades entered by user
53    int gradeCounter; // number of grades entered
54    int grade; // grade value
55    double average; // number with decimal point for average
56
57    // initialization phase
58    total = 0; // initialize total
59    gradeCounter = 0; // initialize loop counter
60
61    // processing phase
62    // prompt for input and read grade from user
63    cout << "Enter grade or -1 to quit: ";
64    cin >> grade; // input grade or sentinel value
65
66    // loop until sentinel value read from user
67    while ( grade != -1 ) // while grade is not -1
68    {
69       total = total + grade; // add grade to total
70       gradeCounter = gradeCounter + 1; // increment counter
71
72       // prompt for input and read next grade from user
73       cout << "Enter grade or -1 to quit: ";
74       cin >> grade; // input grade or sentinel value
75    } // end while
76
77    // termination phase
78    if ( gradeCounter != 0 ) // if user entered at least one grade...
79    {
80       // calculate average of all grades entered
81       average = static_cast< double >( total ) / gradeCounter;
82
83       // display total and average (with two digits of precision)
84       cout << "\nTotal of all " << gradeCounter << " grades entered is "
85          << total << endl;
86       cout << "Class average is " << setprecision( 2 ) << fixed << average
87          << endl;
88    } // end if
89    else // no grades were entered, so output appropriate message
90       cout << "No grades were entered" << endl;
91 } // end function determineClassAverage
```

图 4.13　采用标记控制的循环解决求全班平均成绩的问题：GradeBook 的源代码文件（续）

图4.13中的第55行声明了一个double变量average。记得我们在上一小节的例子中使用一个int变量存储GradeBook类的average。在当前的例子中采用double类型可以使程序在average中存放班级平均成绩计算的浮点数结果。第59行初始化变量gradeCounter为0，因为目前并没有输入任何成绩。请记住，本程序采用的是标记控制的循环。为了准确记录输入的成绩个数，程序只有在用户输入一个合法值（也就是说，不是标记值）时才对变量gradeCounter增1。也就是说，该程序完成对成绩的处理。最后，请注意：两条输入语句（第64和74行）前面都各有一条提示用户输入的输出语句。

```
1  // Fig. 4.14: fig04_14.cpp
2  // Create GradeBook object and invoke its determineClassAverage function.
3
4  // include definition of class GradeBook from GradeBook.h
5  #include "GradeBook.h"
6
7  int main()
8  {
9     // create GradeBook object myGradeBook and
10    // pass course name to constructor
11    GradeBook myGradeBook( "CS101 C++ Programming" );
12
13    myGradeBook.displayMessage(); // display welcome message
14    myGradeBook.determineClassAverage(); // find average of 10 grades
15    return 0; // indicate successful termination
16 } // end main
```

```
Welcome to the grade book for
CS101 C++ Programming

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of all 3 grades entered is 257
Class average is 85.67
```

图4.14 采用标记控制的循环解决求全班平均成绩的问题：创建GradeBook类（见图4.12和图4.13）的一个对象并调用其成员函数determineClassAverage



良好的编程习惯 4.7

请在每次用户输入之前最好给出相应的提示。提示应该指出输入数据的形式和所有的特殊输入值。例如，在一个标记控制的循环中，请求输入的提示应该明确提醒用户标记值是什么。

标记控制的循环和计数器控制的循环蕴涵的程序逻辑

在这里，我们对本节应用程序中标记控制的循环和图4.9中计数器控制的循环，从程序逻辑的角度进行一番比较。在计数器控制的循环中，就指定的迭代次数而言，while语句（见图4.9中的第57~63行）的每次迭代都从用户那里读入一个值。在标记控制的循环中，程序在执行while语句前先读入第一个值（见图4.13中的第63~64行），它决定了该程序的控制流程是否应该进入while循环语句的体内。如果while语句的条件为假，即用户输入的是标记值，那么while循环体不会执行（也就是说，没有成绩输入）。反之，如果条件为真，while循环体开始执行：首先将该成绩值加进total中（第69行），接着循环体中的第73~74行代码从用户那里读入下一个输入的值，然后程序控制到达第75行的代表循环体结束的右花括号（}）。当遇到体结束花括号后，程序继续执行第67行的while循环条件的测试。该条件使用了用户最新输入的成绩，控制循环体是否应该再次执行。请注意，变量grade的值总是在while循环条件测试之前及时地输入。这样，程序可以在处理这个刚刚输入的值（即，将其加到total中并把gradeCounter增1）之前，判定它是否是标记值。如果输入的这个值确实是标记值，那么循环结束，而且程序不会将在此的标记值-1加到total中。

当循环结束时，第78~90行的if...else语句将被执行。第78行的条件确定是否有成绩输入。如果没有，则该if...else语句的else部分（第89~90行）执行，并显示信息“No grades were entered”（未输入任何成绩）。然后，此成员函数将控制权返还给主调函数。

请注意图4.13中while循环内的语句块。如果没有花括号对，循环体中的后面三条语句将落在循环之外，这样造成计算机对这些代码产生如下错误的解释：

Print a summary of the exam results and decide whether tuition should be raised（输出考试结果的总结，并决定是否应该增加学费）

的细化结果是：

Print the number of passes（打印 passes 的值）
Print the number of failures（打印 failures 的值）
If more than eight students passed（如果有 8 各以上的学生通过考试）
　Print “Raise tuition”（打印“Raise tuition”）

图 4.15 是对上述两级细化结果的汇总。请注意，为了提高可读性，其中的 While 结构仍用空行与其他语句分开。现在列出的伪代码，其详细程度已经达到可以转换为 C++ 程序的要求了。

```
1  Initialize passes to zero
2  Initialize failures to zero
3  Initialize student counter to one
4
5  While student counter is less than or equal to 10
6     Prompt the user to enter the next exam result
7     Input the next exam result
8
9     If the student passed
10       Add one to passes
11    Else
12       Add one to failures
13
14    Add one to student counter
15
16 Print the number of passes
17 Print the number of failures
18
19 If more than eight students passed
20    Print "Raise tuition"
```

图 4.15　考试结果问题的伪代码

转化到 Analysis 类

图 4.16 和图 4.17 给出了实现图 4.15 中伪代码算法的 C++ 类，图 4.18 则给出了该类的两个运行示例。

```
1  // Fig. 4.16: Analysis.h
2  // Definition of class Analysis that analyzes examination results.
3  // Member function is defined in Analysis.cpp
4
5  // Analysis class definition
6  class Analysis
7  {
8  public:
9     void processExamResults(); // process 10 students' examination results
10 }; // end class Analysis
```

图 4.16　考试结果问题：Analysis 类的头文件

图 4.17 中的第 16~18 行声明了 Analysis 类的成员函数 processExamResults 的变量，这个函数是用来处理考试结果的。请注意，这里我们利用了 C++ 声明的同时可以初始化的特点（passes 初始化

为0，failures初始化为0，以及studentCounter初始化为1）。循环程序可能需要在每次循环开始时初始化变量。这种重新初始化的实现方式通常有两种：一是用赋值语句，而非声明语句；二是将声明语句挪至循环体内。

```
1  // Fig. 4.17: Analysis.cpp
2  // Member-function definitions for class Analysis that
3  // analyzes examination results.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  // include definition of class Analysis from Analysis.h
10 #include "Analysis.h"
11
12 // process the examination results of 10 students
13 void Analysis::processExamResults()
14 {
15    // initializing variables in declarations
16    int passes = 0; // number of passes
17    int failures = 0; // number of failures
18    int studentCounter = 1; // student counter
19    int result; // one exam result (1 = pass, 2 = fail)
20
21    // process 10 students using counter-controlled loop
22    while ( studentCounter <= 10 )
23    {
24       // prompt user for input and obtain value from user
25       cout << "Enter result (1 = pass, 2 = fail): ";
26       cin >> result; // input result
27
28       // if...else nested in while
29       if ( result == 1 )           // if result is 1,
30          passes = passes + 1;      // increment passes;
31       else                         // else result is not 1, so
32          failures = failures + 1; // increment failures
33
34       // increment studentCounter so loop eventually terminates
35       studentCounter = studentCounter + 1;
36    } // end while
37
38    // termination phase; display number of passes and failures
39    cout << "Passed " << passes << "\nFailed " << failures << endl;
40
41    // determine whether more than eight students passed
42    if ( passes > 8 )
43       cout << "Raise tuition " << endl;
44 } // end function processExamResults
```

图 4.17　考试结果问题：Analysis类源代码文件中的嵌套的控制语句

第22~36行的while循环语句共迭代十次。在每一次迭代期间，这条循环语句输入并处理一个考试结果。请注意，处理每个考试结果的if...else（第29~32行）嵌套在while循环语句体中。如果结果是1，if...else语句将passes增1；否则，它假定结果为2，并对failures增1。在第22行的循环条件被再次测试之前，第35行将studentCounter增1。当输入10个值后，循环结束。第39行的代码执行后将输出passes和failures的值。第42~43行的if语句判断是否有8个以上的学生通过考试，如果条件为真，则输出信息“Raise Tuition”。

```
1  // Fig. 4.18: fig04_18.cpp
2  // Test program for class Analysis.
3  #include "Analysis.h" // include definition of class Analysis
4
5  int main()
6  {
7     Analysis application; // create Analysis object
8     application.processExamResults(); // call function to process results
9     return 0; // indicate successful termination
10 } // end main
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed 9
Failed 1
Raise tuition
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Passed 6
Failed 4
```

图 4.18　Analysis 类的测试程序

Analysis 类的运行演示

图 4.18 中的第 7 行创建了 Analysis 类的一个对象，并调用其成员函数 processExamResults（第 8 行）来处理用户输入的一组考试结果。图 4.18 显示了该程序两次运行示例的输入和输出。在第一次运行结束时，图 4.17 中的成员函数 processExamResults 的条件（第 42 行）为真——超过 8 个学生通过了考试，因此程序输出学费应该增加的一条信息。

4.11　赋值运算符

C++ 提供了多种赋值运算符来缩写赋值表达式。例如，下面的语句：

```
c = c + 3;
```

可以用加法赋值运算符 += 将其缩写为：

```
c += 3;
```

+= 运算符把运算符右边表达式的值和运算符左边的变量的值相加，结果存放在运算符左边的变量中。任何如下形式的语句：

的头文件和源代码文件），以及测试这个类的另一个源代码文件。该测试用的源代码文件包含创建类的一个对象和调用其成员函数的main函数。在图4.21的例子中，我们只是想说明++运算符的技术细节，所以仅仅用了一个包含main函数的源代码文件。有时候，当创建一个可重用的类来演示一个简单的概念没有多大意义时，我们将用一个只包含所有功能代码的main函数的源代码文件来实现。

```
1  // Fig. 4.21: fig04_21.cpp
2  // Preincrementing and postincrementing.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int c;
10
11    // demonstrate postincrement
12    c = 5; // assign 5 to c
13    cout << c << endl; // print 5
14    cout << c++ << endl; // print 5 then postincrement
15    cout << c << endl; // print 6
16
17    cout << endl; // skip a line
18
19    // demonstrate preincrement
20    c = 5; // assign 5 to c
21    cout << c << endl; // print 5
22    cout << ++c << endl; // preincrement then print 6
23    cout << c << endl; // print 6
24    return 0; // indicate successful termination
25 } // end main
```

```
5
5
6

5
6
6
```

图4.21 前置自增和后置自增

第12行将变量c初始化为5，第13行输出c的初始值。第14行输出表达式c++的值。这个表达式后置自增变量c，所以输出c的初始值（5），然后c的值增1。因此，第14行又一次输出c的初始值（5）。第15行输出了c的新值（6），证明在第14行该变量的值的确增加了1。

第20行重设c的值为5，第21行输出c的值。第22行输出表达式++c的值。该表达式前置自增c，因此它的值先增1，然后输出这个新值（6）。第23行又一次输出c的值，证明在第22行被执行后c的值仍然是6。

算术赋值运算符、自增和自减运算符可以用来简化程序语句。图4.17中的3条赋值语句：

```
passes = passes + 1 ;
failures = failures + 1 ;
studentCounter = studentCounter + 1 ;
```

可以用算术赋值运算符精简为：

b) `cin << value;`

c)
```
if ( gender == 1 )
   cout << "Woman" << endl;
else;
   cout << "Man" << endl;
```

4.10 请指出下面while循环语句中存在的问题。

```
while ( z >= 0 )
   sum += z;
```

自测题答案

4.1 a) 顺序、选择和循环。b) if...else。c) 计数器控制的或定数的。d) 哨兵/信号/标记/哑元

4.2
```
x = x + 1;
x += 1;
++x;
x++;
```

4.3 a) `z = x++ + y;`

b)
```
if ( count > 10 )
   cout << "Count is greater than 10" << endl;
```

c) `total -= --x;`

d) `q %= divisor;`
 `q = q % divisor;`

4.4 a)
```
int sum;
int x;
```

b) `x = 1 ;`

c) `sum = 0 ;`

d)
```
sum += x;
```
或
```
sum = sum + x;
```

e) `cout << "The sum is: " << sum << endl;`

4.5 请看下面的代码：

```
// Exercise 4.5 Solution: ex04_05.cpp
// Calculate the sum of the integers from 1 to 10.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
   int sum; // stores sum of integers 1 to 10
   int x; // counter

   x = 1; // count from 1
   sum = 0; // initialize sum

   while ( x <= 10 ) // loop 10 times
   {
      sum += x; // add x to sum
```

```
18          x++; // increment x
19       } // end while
20
21       cout << "The sum is: " << sum << endl;
22       return 0; // indicate successful termination
23    } // end main
```

```
The sum is: 55
```

4.6 a) `product = 25, x = 6;`

b) `quotient = 0 , x = 6;`

```
1  // Exercise 4.6 Solution: ex04_06.cpp
2  // Calculate the value of product and quotient.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 5;
10    int product = 5;
11    int quotient = 5;
12
13    // part a
14    product *= x++; // part a statement
15    cout << "Value of product after calculation: " << product << endl;
16    cout << "Value of x after calculation: " << x << endl << endl;
17
18    // part b
19    x = 5; // reset value of x
20    quotient /= ++x; // part b statement
21    cout << "Value of quotient after calculation: " << quotient << endl;
22    cout << "Value of x after calculation: " << x << endl << endl;
23    return 0; // indicate successful termination
24 } // end main
```

```
Value of product after calculation: 25
Value of x after calculation: 6

Value of quotient after calculation: 0
Value of x after calculation: 6
```

4.7 a) `cin >> x;`

b) `cin >> y;`

c) `i = 1;`

d) `power = 1;`

e) `power *= x;`

或

`power = power * x;`

f) `i++;`

g) `if ( i <= y )`

h) `cout << power << endl;`

4.8 请看下面的代码：

```
// Exercise 4.8 Solution: ex04_08.cpp
// Raise x to the y power.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main()
{
   int x; // base
   int y; // exponent
   int i; // counts from 1 to y
   int power; // used to calculate x raised to power y

   i = 1; // initialize i to begin counting from 1
   power = 1; // initialize power

   cout << "Enter base as an integer: ";  // prompt for base
   cin >> x; // input base

   cout << "Enter exponent as an integer: "; // prompt for exponent
   cin >> y; // input exponent

   // count from 1 to y and multiply power by x each time
   while ( i <= y )
   {
      power *= x;
      i++;
   } // end while

   cout << power << endl; // display result
   return 0; // indicate successful termination
} // end main
```

```
Enter base as an integer: 2
Enter exponent as an integer: 3
8
```

4.9 a) 错误：缺少 while 体的结束右花括号。

改正：在语句 c++;后添加结束右花括号。

b) 错误：使用流插入运算符，而非流提取运算符。

改正：将<<改成>>。

c) 错误：else 之后的分号导致了一个逻辑错误，使得第二条输出语句始终都会执行。

改正：删除 else 之后的分号。

4.10 变量 z 的值在 while 语句执行过程中永远不会改变。因此，如果循环继续的条件(z >= 0)最初为真的话，就会产生无限循环。为了避免无限循环，z 的值必须自减，使其最终能够小于 0。

习题

4.11 指出并改正以下各代码段中的错误：

a)
```
if ( age >= 65 );
   cout << "Age is greater than or equal to 65" << endl;
else
   cout << "Age is less than 65 << endl";
```

b)
```
if ( age >= 65 )
   cout << "Age is greater than or equal to 65" << endl;
else;
   cout << "Age is less than 65 << endl";
```

c)
```
int x = 1, total;

while ( x <= 10 )
{
   total += x;
   x++;
}
```

d)
```
While ( x <= 100 )
   total += x;
   x++;
```

e)
```
while ( y > 0 )
{
   cout << y << endl;
   y++;
}
```

4.12 请写出下面程序的输出结果：

```
// Exercise 4.12: ex04_12.cpp
// What does this program print?
#include <iostream>
using std::cout;
using std::endl;

int main()
{
   int y; // declare y
   int x = 1; // initialize x
   int total = 0; // initialize total

   while ( x <= 10 ) // loop 10 times
   {
      y = x * x; // perform calculation
      cout << y << endl; // output result
      total += y; // add y to total
      x++; // increment counter x
   } // end while

   cout << "Total is " << total << endl; // display result
   return 0; // indicate successful termination
} // end main
```

对于习题4.13到习题4.16，请执行以下步骤：

a) 阅读问题陈述。

b) 使用伪代码和自顶向下逐步求精法制定算法。

c) 编写C++程序。

d) 测试、调试和执行该C++程序。

4.13 每位司机都关心自己车辆的行车里程数。有位司机通过记录每加一次油所行驶的英里数和油的加仑数来跟踪他的加油情况。请开发一个C++程序，它使用一条while语句输入

每次加油行驶英里数和加油量。该程序应计算和显示每次加油所得到的每加仑行驶英里数，并打印到本次加油，所有加油综合的每加仑英里数。

```
Enter the miles used (-1 to quit): 287
Enter gallons: 13
MPG this tankful: 22.076923
Total MPG: 22.076923

Enter the miles used (-1 to quit): 200
Enter gallons: 10
MPG this tankful: 20.000000
Total MPG: 21.173913

Enter the miles used (-1 to quit): 120
Enter gallons: 5
MPG this tankful: 24.000000
Total MPG: 21.678571

Enter miles (-1 to quit): -1
```

4.14 开发一个C++程序，它可以判断商场顾客的支付款额是否超出了赊欠账户的信用额度。对每位顾客，提供如下的信息：

a) 账号（是一个整数）；

b) 月初欠款；

c) 本月该顾客购买的所有商品的总金额；

d) 本月该顾客账户存入的总金额；

e) 允许的信用额度。

这一程序应使用一条while语句输入以上的每条信息，计算新的欠款（=月初欠款+当月消费额-银行存款），并判断新的欠款是否超过客户的信用额度。对那些已经超支的顾客，程序应显示他的账号、信用额度、新的欠款以及一条消息“Credit Limit Exceeded.”。

```
Enter account number (-1 to end): 100
Enter beginning balance: 5394.78
Enter total charges: 1000.00
Enter total credits: 500.00
Enter credit limit: 5500.00
New balance is 5894.78
Account:      100
Credit limit: 5500.00
Balance:      5894.78
Credit Limit Exceeded.

Enter Account Number (or -1 to quit): 200
Enter beginning balance: 1000.00
Enter total charges: 123.45
Enter total credits: 321.00
Enter credit limit: 1500.00
New balance is 802.45

Enter Account Number (or -1 to quit): 300
Enter beginning balance: 500.00
Enter total charges: 274.73
Enter total credits: 100.00
Enter credit limit: 800.00
New balance is 674.73

Enter Account Number (or -1 to quit): -1
```

4.15 一家大型化工厂采用佣金方式为推销员付酬金。推销员每周领到基本工资200美元，再加上这周销售毛利的9%。例如，某位销售员某周卖出了价值5000美元的化工产品，那么除领取基本的200美元外，还有5000美元的9%，总计650美元。开发一个C++程序，它利用一条while语句输入每位销售员上周的毛利，然后计算和显示其收入。每次处理一个销售员的数据。

```
Enter sales in dollars (-1 to end): 5000.00
Salary is: $650.00

Enter sales in dollars (-1 to end): 6000.00
Salary is: $740.00

Enter sales in dollars (-1 to end): 7000.00
Salary is: $830.00

Enter sales in dollars (-1 to end): -1
```

4.16 开发一个C++程序，它利用一条while语句决定每位雇员的薪金总额。公司规定，每位雇员在其工作的前40个小时领取“正规工作时间”工资，超过40小时后，领取“相当于原工资一倍半的加班费”工资。现在，给你公司中所有雇员的一份工作清单，上列每位雇员上周工作的小时数和“正规工作时间”每小时的工资数。要求编写的程序能够输入每位雇员的这些信息，并确定和显示雇员上周的薪金总额。

```
Enter hours worked (-1 to end): 39
Enter hourly rate of the worker ($00.00): 10.00
Salary is $390.00

Enter hours worked (-1 to end): 40
Enter hourly rate of the worker ($00.00): 10.00
Salary is $400.00

Enter hours worked (-1 to end): 41
Enter hourly rate of the worker ($00.00): 10.00
Salary is $415.00

Enter hours worked (-1 to end): -1
```

4.17 在计算机应用中，我们常常会遇到寻找最大数（即一组数中的最大值）的问题。例如，一个确定一次销售竞赛优胜者的程序，它输入每个销售员的销售量，销售量最多的销售员将赢得这次竞赛的胜利。请首先写出伪代码程序，然后用C++程序实现。要求C++程序通过一条while语句判定和打印用户输入的10个数中的最大数。你的程序应用到下面的3个变量：

counter:	能计数到10的计数器（即用于记录已输入数的个数，并用于判断何时处理完了10个数）；
number:	当前输入到程序的数；
largest:	迄今为止找到的最大数。

4.18 编写一个C++程序，使用一条while语句和制表符的转义序列“\t”，打印下表：

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

4.19 采用类似于习题4.17的方法，从10个数中寻找最大的两个值。注意：每个数只能输入一次。

4.20 图4.16至图4.18给出的考试结果程序假定用户输入的所有值不是1，就是2。请修改这个应用程序，让它满足输入的有效性确认。对于任何一次输入，如果键入的值不是1或2，那么程序保持循环直到用户键入一个正确的值为止。

4.21 请写出下面程序的输出结果：

```
// Exercise 4.21: ex04_21.cpp
// What does this program print?
#include <iostream>
using std::cout;
using std::endl;

int main()
{
   int count = 1; // initialize count

   while ( count <= 10 ) // loop 10 times
   {
      // output line of text
      cout << ( count % 2 ? "****" : "++++++++" ) << endl;
      count++; // increment count
   } // end while

   return 0; // indicate successful termination
} // end main
```

4.22 请写出下面程序的输出结果：

```
// Exercise 4.22: ex04_22.cpp
// What does this program print?
#include <iostream>
using std::cout;
using std::endl;

int main()
{
   int row = 10; // initialize row
   int column; // declare column

   while ( row >= 1 ) // loop until row < 1
   {
      column = 1; // set column to 1 as iteration begins

      while ( column <= 10 ) // loop 10 times
      {
         cout << ( row % 2 ? "<" : ">" ); // output
         column++; // increment column
      } // end inner while

      row--; // decrement row
      cout << endl; // begin new output line
   } // end outer while

   return 0; // indicate successful termination
} // end main
```

4.23（else摇摆问题）在x等于9，y等于11，以及x等于11，y等于9这两种情况下，请说出下面程序段的输出。请注意，编译器将忽略C++程序中的缩进格式。C++编译器总是将else与上一个if联系起来，除非用添加花括号{}的方法另外指定。乍看上去，程序员可能无法确定else与哪一个if匹配，所以称这种情况为“else摇摆”问题。在此，我们为

了增加这个问题的挑战性，而故意消除了下面代码的缩进格式。提示：请应用所学的缩进约定。

a)
```
if ( x < 10 )
if ( y > 10 )
cout << "*****" << endl;
else
cout << "#####" << endl;
cout << "$$$$$" << endl;
```

b)
```
if ( x < 10 )
{
if ( y > 10 )
cout << "*****" << endl;
}
else
{
cout << "#####" << endl;
cout << "$$$$$" << endl;
}
```

4.24（另一个else摇摆问题）修改以下代码，以产生指定的输出。请使用正确的缩进技巧。请注意，除了可以插入花括号外，程序不得做任何其他改动。编译器会忽略C++程序中的缩进格式。同上题，为了增加问题难度，我们故意消除了下面代码的缩进格式。注意：有可能无需做任何修改。

```
if ( y == 8 )
if ( x == 5 )
cout << "@@@@@" << endl;
else
cout << "#####" << endl;
cout << "$$$$$" << endl;
cout << "&&&&&" << endl;
```

a) 假定x = 5，y = 8, 产生的输出如下所示：

```
@@@@@
$$$$$
&&&&&
```

b) 假定x = 5，y = 8, 产生的输出如下所示：

```
@@@@@
```

c) 假定x = 5，y = 8, 产生的输出如下所示：

```
@@@@@
&&&&&
```

d) 假定x = 5，y = 7, 产生的输出如下所示：（注意，else之后的3条输出语句属于同一个语句块。）

```
#####
$$$$$
&&&&&
```

4.25 编写一个程序，它能够读入一个正方形的边长，然后打印出一个由星号和空格组成的、边长为刚读入边长的空心正方形。该程序应可以处理边长在1到20之间的所有正方形。例如，程序读入的边长是5，那么它应打印出如下的空心正方形：

```
*****
*   *
*   *
*   *
*****
```

4.26 所谓“回文”是一种特殊的数或者文字短语。它们无论是顺读，还是倒读，结果都一样。例如，以下的几个5位整数都是回文数：12321、55555、45554和11611。编写一个程序，读入一个5位整数后，能够判断它是否为回文数。提示：除法和取模运算符可以将数分解为单个的个位数字。

4.27 输入仅由0和1构成的整数（也就是“二进制”整数），打印出该数对应的十进制整数。通过运用取模和除法运算符，就可以从这个“二进制”整数中，按从右至左的顺序，每次“剔”出一个“二进制”位的数字。我们知道，在十进制计数体制中，最右数字的位置值为1，然后向左的数字依次的位置值是10、100、1000等。同理，在二进制计数体制中，最右数字的位置值为1，然后向左的数字依次的位置值是2、4、8，等等。所以，二进制数234可以理解成：2 * 100 + 3 * 10 + 4 * 1，而与二进制数1101等价的十进制数是1 * 1 + 0 * 2 + 1 * 4 + 1 * 8或1 + 0 + 4 + 8，也就是13。注意：对二进制数不熟悉的读者可以参考附录D。

4.28 编写一个程序，显示如下所示的棋盘式图案。要求你的程序只能使用以下3条输出语句。

```
cout << "* ";
cout << ' ';
cout << endl;
```

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

4.29 编写一个程序，能够连续打印出整数2的幂，即2、4、8、16、32、64.....在这里，要求你的程序中的while循环不能终止（也就是，你得创建一个无限的循环）。为此，可以简单地将关键字true作为这条while语句的继续条件表达式。那么，运行这个程序会发生什么情况？

4.30 编写一个程序，读入圆的半径（double类型的值）后，计算并打印直径、周长和面积。要求采用的π值是3.14159。

和原则。本章和第4章所讲述的这些控制语句将帮助大家构建和操纵对象。本书从一开始就强调要尽早进入面向对象编程。首先在第 1 章中对基本概念进行了讨论，然后在第 3 和 4 章中通过许多面向对象代码的例子进行举例说明，并给出了大量的习题。在这里，我们继续将重点放在面向对象编程的介绍上。

本章展示 for 语句、do...while 语句和 switch 语句。通过一系列使用 while 语句和 for 语句的小例子，考察计数器控制的循环的要素。这一章部分篇幅继续扩展在第 3 和 4 章呈现的 GradeBook 类。特别是，我们创建的 GradeBook 类的这个版本，使用 switch 语句统计用户输入的一组字母打分成绩中 A、B、C、D 和 F 等级的各自数量。本章介绍 break 和 continue 这两个程序控制语句，讨论逻辑运算符，后者使程序员可以在控制语句中使用更强大的条件表达式。我们也分析一个常见的错误，即混淆相等运算符（==）和赋值运算符（=），并研究如何避免的对策。最后，我们对于本章和第4章中介绍的 C++ 控制语句和已证明的问题解决技术做了总结。

5.2　计数器控制的循环的要素

本节使用第 4 章中介绍的 while 循环语句，正式地介绍执行计数器控制的循环所要求的元素。计数器控制的循环需要：

1. 命名一个控制变量（或者循环计数器）。
2. 设置控制变量的初值。
3. 定义一个循环继续条件，用于对控制变量终值的测试（例如，循环是否应该继续）。
4. 增值（或减值），即在每次循环过程中修改控制变量的值。

不妨考虑图 5.1 中的这个简单程序，它打印从 1 到 10 的数字。第 9 行的声明是对控制变量（counter）的命名，声明它是一个整型变量，为其在内存中预留空间，并设置它的初值为1。要求初始化的声明从效果上讲是可执行的语句。在C++中，正如刚才的声明所为，对同时预留内存空间的声明（declaration）来说，其更确切的叫法应该是定义（definition）。由于定义也是声明，因此除非这种区别特别重要，一般情况下我们还是使用术语“声明”。

```
1  // Fig. 5.1: fig05_01.cpp
2  // Counter-controlled repetition.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int counter = 1; // declare and initialize control variable
10
11    while ( counter <= 10 ) // loop-continuation condition
12    {
13       cout << counter << " ";
14       counter++; // increment control variable by 1
15    } // end while
16
17    cout << endl; // output a newline
18    return 0; // successful termination
19 } // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

图 5.1　计数器控制的循环

```
1  // Fig. 5.2: fig05_02.cpp
2  // Counter-controlled repetition with the for statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     // for statement header includes initialization,
10    // loop-continuation condition and increment.
11    for ( int counter = 1; counter <= 10; counter++ )
12       cout << counter << " ";
13
14    cout << endl; // output a newline
15    return 0; // indicate successful termination
16 } // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

图5.2　使用for语句的计数器控制的循环

for语句头部的构成成分

图5.3对图5.2的for语句头部（第11行）进行了更进一步的说明。请注意，这条for语句的头部“做了所有该做的事情”——它利用一个控制变量指定了计数器控制的循环所需的每一项。如果for语句循环体中的体语句超过一条的话，那么就要求用一对花括号括住它们，形成循环体。

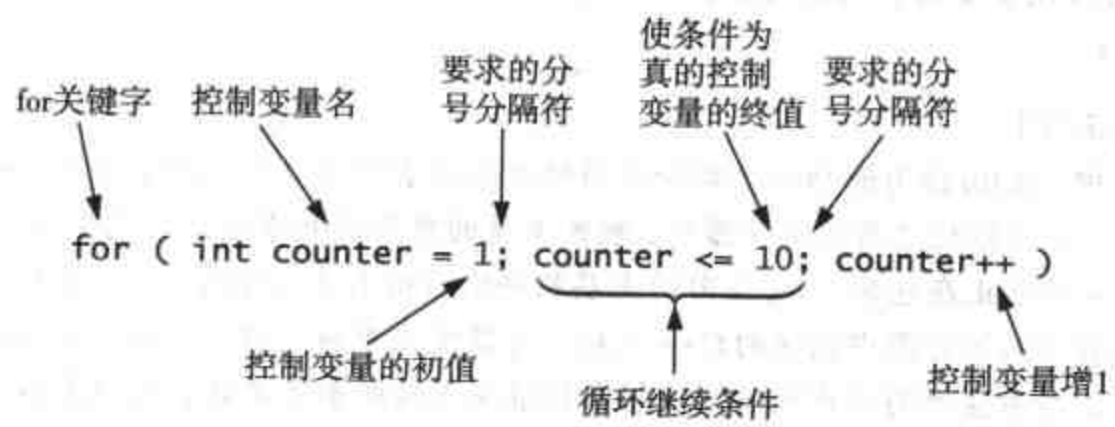


图5.3　for语句头部的组成

注意，图5.2使用的循环继续条件是counter <= 10。假如程序员错误地将其写成counter < 10，那么循环体就只执行9次。这是一个常见的相差1的错误。



常见的编程错误 5.2

在while语句或者for语句的条件中，如果使用了不正确的关系运算符，或者使用了不正确的循环计数器终值，会导致相差1的错误。



良好的编程习惯 5.4

在while语句或者for语句的条件中使用终值，并使用关系运算符<=，有助于避免相差1的错误。例如，对于打印值1到10这样的循环，循环继续条件应是counter <= 10，而不应该是counter < 10（这是一个相差1的错误）或者counter < 11（虽然这是正确的）。为了达到循环计数10次的目的，许多程序员更喜欢所谓的“基于0的计数法”，也就是，将循环语句的控制变量counter初始化为0，并将循环继续的测试条件设置为counter < 10。

for语句的一般形式是：

语句的UML活动图。程序员的任务便是采用堆叠和嵌套的方式，将这些for语句活动图同其他控制语句的活动图组合起来，以形成算法的结构化实现。然后，程序员再用适合于算法的动作表达式和监控条件填写动作状态和判定符号。

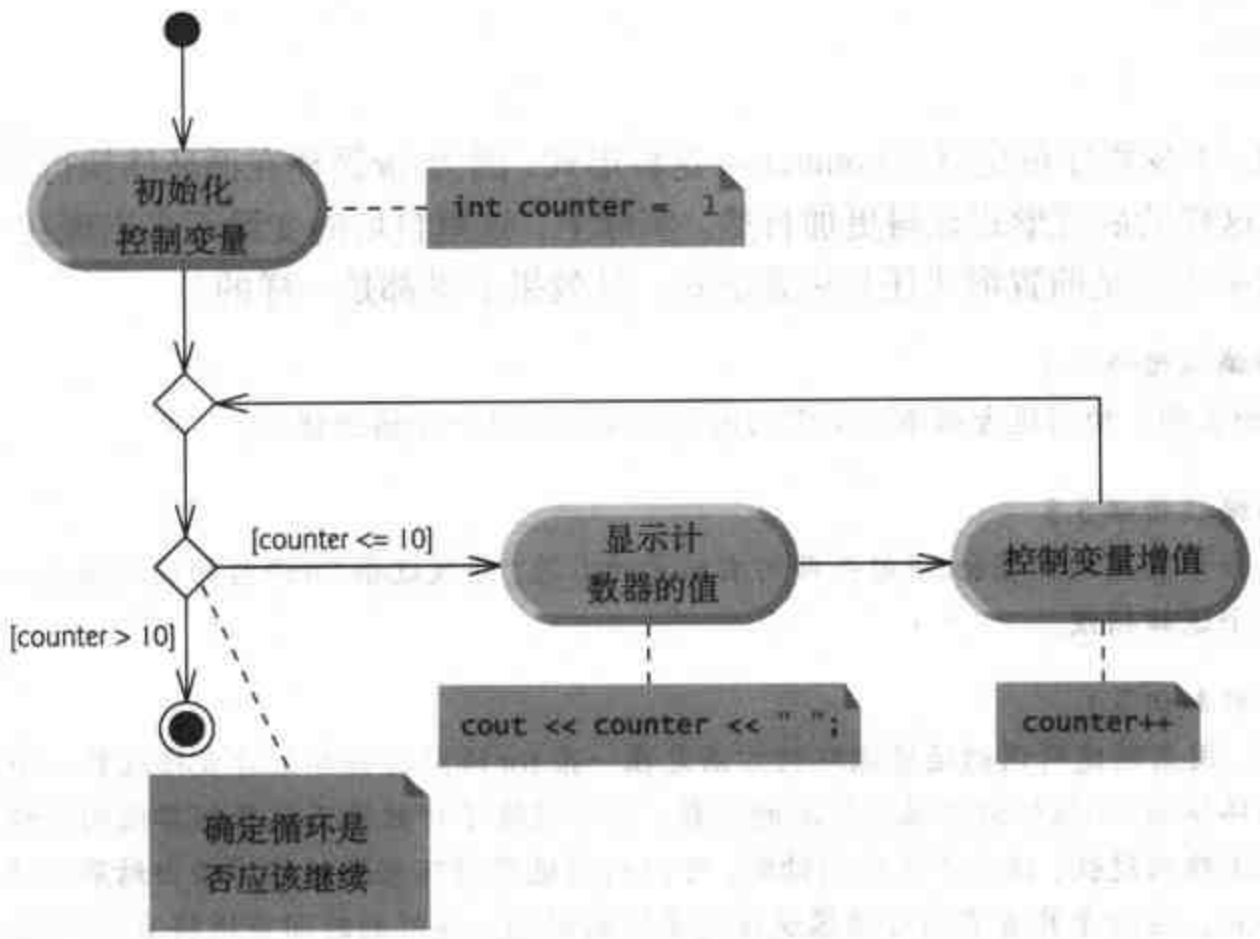


图 5.4 图 5.2 中 for 语句的 UML 活动图

5.4 使用 for 语句的例子

下面的例子展示了在for语句中改变控制变量的多种方法。在每个例子中，我们都编写了一个适当的 for 语句头部。请注意，为递减控制变量的循环所提供的关系运算符的变化。

a) 控制变量从 1 变到 100，每次增量（或步长）为 1。

```
for ( int i = 1 ; i <= 100; i++ )
```

b) 控制变量从 100 变到 1，每次增量（或步长）为 -1（就是减量为 1）。

```
for ( int i = 100; i >= 1 ; i-- )
```

c) 控制变量从 7 变到 77，每次增量（或步长）为 7。

```
for ( int i = 7 ; i <= 77; i += 7 )
```

d) 控制变量从 20 变到 2，每次增量（或步长）为 -2。

```
for ( int i = 20; i >= 2 ; i -= 2 )
```

e) 控制变量的值依次变化的序列是：2、5、8、11、14、17、20。

```
for ( int i = 2 ; i <= 20; i += 3 )
```

f) 控制变量的值依次变化的序列是：99、88、77、66、55、44、33、22、11、0。

```
for ( int i = 99; i >= 0 ; i -= 11 )
```



常见的编程错误 5.6

如果在实现倒数计数的循环继续条件中使用了不正确的关系运算符，例如，在一个倒数计数到1的循环中错误地使用了i<=1而不是i>=1，通常是一个使程序可以运行、但产生不正确结果的逻辑错误。

应用：计算从2到20所有偶数的和

接下来，我们通过两个例子说明如何用for语句解决简单的应用问题。图5.5的程序使用一条for语句计算从2到20所有偶数的和。第12~13行的这条循环语句在每次迭代时都将控制变量number的当前值累加到变量total中。

```
1  // Fig. 5.5: fig05_05.cpp
2  // Summing integers with the for statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int total = 0; // initialize total
10
11    // total even integers from 2 through 20
12    for ( int number = 2; number <= 20; number += 2 )
13       total += number;
14
15    cout << "Sum is " << total << endl; // display results
16    return 0; // successful termination
17 } // end main
```

```
Sum is 110
```

图5.5　用for语句计算整数的和

请注意，通过使用逗号运算符，图5.5中for语句的循环体实际上可以合并到for头部的增值部分，结果如下所示：

```
for ( int number = 2 ; // initialization
      number <= 20; // loop continuation condition
      total += number, number += 2 ) // total and increment
   ; // empty body
```



良好的编程习惯 5.6

虽然在for语句前面的语句和for的循环体内的语句常常可以合并到for语句的头部中，但是这样做往往会增加阅读、维护、修改和调试程序的难度。



良好的编程习惯 5.7

如果可能的话，尽量将控制语句的头部规模限制在一行内。

应用：计算复利

下面的例子使用一条for语句计算复利（所谓复利是指以本金加上未付的利息为基数计算的利息）。仔细考虑下面的问题陈述：

> 某人在一个年利率是5%的储蓄账户中存入$1000.00。假定所有的利息都重复存入账户，请计算并打印在10年中每年年终时此账户中的存款金额。这些金额的计算公式如下：
>
> $a = p(1+r)^n$
>
> 其中，p代表最初存入的金额（也就是本金），r代表年利率，n代表年数，a代表第n年年终的存款金额。

这个问题的解决涉及到了循环，这里的循环对于10年中的每一年都要执行指定的获得存款金额的计算。该问题的解答如图5.6所示。

```
1  // Fig. 5.6: fig05_06.cpp
2  // Compound interest calculations with for.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setw; // enables program to set a field width
10 using std::setprecision;
11
12 #include <cmath> // standard C++ math library
13 using std::pow; // enables program to use function pow
14
15 int main()
16 {
17    double amount; // amount on deposit at end of each year
18    double principal = 1000.0; // initial amount before interest
19    double rate = .05; // interest rate
20
21    // display headers
22    cout << "Year" << setw( 21 ) << "Amount on deposit" << endl;
23
24    // set floating-point number format
25    cout << fixed << setprecision( 2 );
26
27    // calculate amount on deposit for each of ten years
28    for ( int year = 1; year <= 10; year++ )
29    {
30       // calculate new amount for specified year
31       amount = principal * pow( 1.0 + rate, year );
32
33       // display the year and the amount
34       cout << setw( 4 ) << year << setw( 21 ) << amount << endl;
35    } // end for
36
37    return 0; // indicate successful termination
38 } // end main
```

```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.63
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

图5.6 使用for语句的复合利息计算

第28~35行的for语句共执行它的循环体10次，控制变量从1变化到10，每次的增量为1。C++并没有求幂的运算符，所以我们利用了能完成求幂任务的标准库函数pow（第31行）。函数pow(x,y)计算x的y次幂的值。在这个例子中，代数式$(1 + r)^n$写成pow（1.0 + rate，year），其中变量rate表示r，变量year表示n。函数pow需要接受两个类型为double的实参，并返回一个double类型的值。

```
1  // Fig. 5.7: fig05_07.cpp
2  // do...while repetition statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int counter = 1; // initialize counter
10
11    do
12    {
13       cout << counter << " "; // display counter
14       counter++; // increment counter
15    } while ( counter <= 10 ); // end do...while
16
17    cout << endl; // output a newline
18    return 0; // indicate successful termination
19 } // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

图5.7　do...while循环语句

do...while语句的UML活动图

图5.8包含了do...while语句的UML活动图。这张图清楚地显示出循环继续条件直到循环语句的循环体至少执行一次后，才被测试。将此张活动图与图4.6中的while语句的活动图做比较，你会再次注意到，这张活动图同样只包含若干动作状态和一个判定符号（除了一个初始状态、多个转换箭头、一个合并符号、一个结束状态和几个注释之外）。请大家再次想像一下，程序员有一个大柜子，其中包含了大量足够他使用的、空的do...while语句的UML活动图。程序员的任务便是采用堆叠和嵌套的方式，将这些do...while语句活动图同其他控制语句的活动图组合起来，以形成算法的结构化实现。然后，程序员再用适合于算法的动作表达式和监控条件填写动作状态和判定符号。

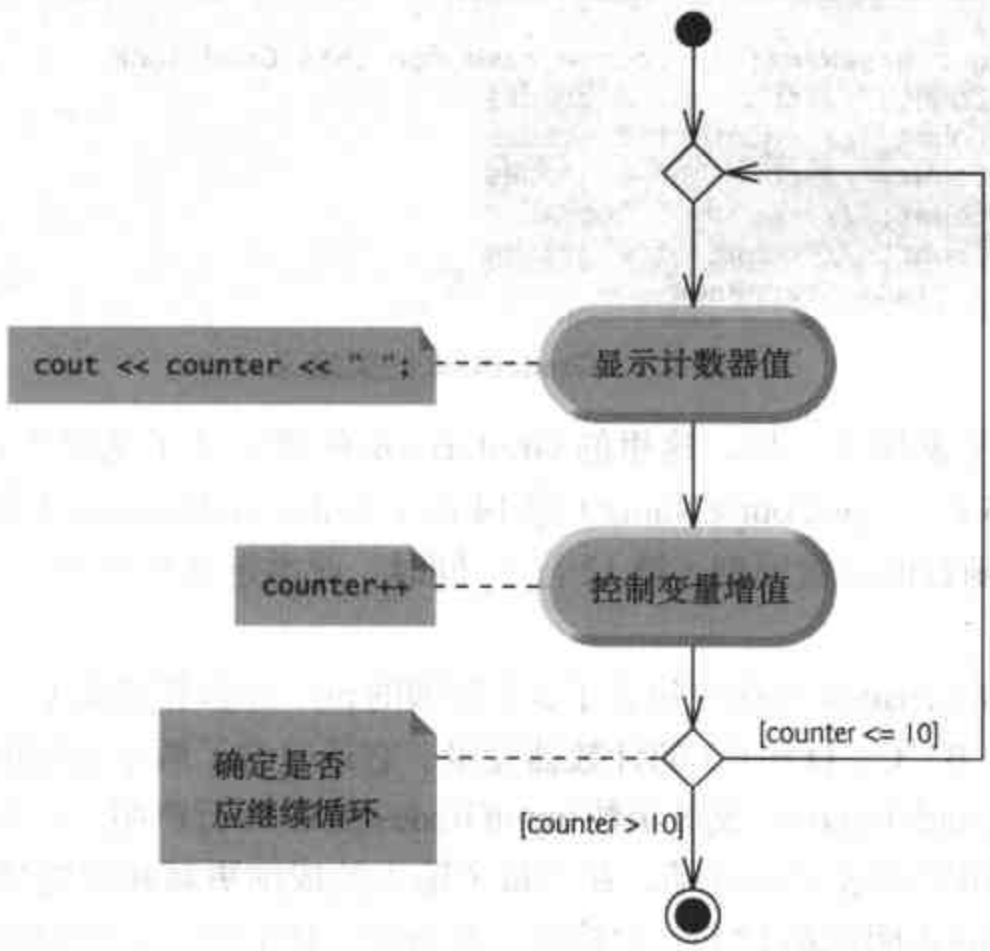


图5.8　图5.7的do...while循环语句的UML活动图

5.6 switch 多路选择语句

我们在第 4 章讨论了 if 单路选择语句和 if...else 双路选择语句，不过 C++ 还提供了 switch 多路选择语句，它根据一个变量或表达式可能发生的值实施不同的动作。每个动作都跟一个常整数表达式（即字符常量和整数常量的任意组合，其求值结果是一个常整数值）的值相关联，并且这个常整数表达式的值正是 switch 语句执行所依据的那个变量或表达式可能采取的。

使用 switch 语句统计 A、B、C、D 和 F 级成绩的 GradeBook 类

在接下来的例子中，我们给出 GradeBook 类的一个增强版本，即这个 GradeBook 类是对第 3 章引入的、第 4 章进一步开发的 GradeBook 类的继续完善。这一新版本的类要求用户输入一系列用字母表示的成绩，然后输出每级成绩对应的学生数的一份总结。该类使用一条 switch 语句判断输入的成绩级别是否是 A、B、C、D 和 F，并使相应的成绩计数器自增。GradeBook 类定义在图 5.9 中，而它的成员函数的定义出现在图 5.10 中。图 5.11 针对使用 GradeBook 类进行一组成绩处理的 main 函数，显示其执行时输入和输出的样例。

```
1  // Fig. 5.9: GradeBook.h
2  // Definition of class GradeBook that counts A, B, C, D and F grades.
3  // Member functions are defined in GradeBook.cpp
4
5  #include <string> // program uses C++ standard string class
6  using std::string;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    GradeBook( string ); // constructor initializes course name
13    void setCourseName( string ); // function to set the course name
14    string getCourseName(); // function to retrieve the course name
15    void displayMessage(); // display a welcome message
16    void inputGrades(); // input arbitrary number of grades from user
17    void displayGradeReport(); // display a report based on the grades
18 private:
19    string courseName; // course name for this GradeBook
20    int aCount; // count of A grades
21    int bCount; // count of B grades
22    int cCount; // count of C grades
23    int dCount; // count of D grades
24    int fCount; // count of F grades
25 }; // end class GradeBook
```

图 5.9 GradeBook 的类定义

与这个类定义的早期版本一样，这里的 GradeBook 的类定义（见图 5.9）既包含了成员函数 setCourseName（第 13 行）、getCourseName（第 14 行）和 displayMessage（第 15 行）的函数原型，又包含了该类的构造函数的函数原型（第 12 行）。同时，此类定义在第 19 行也声明了 private 数据成员 courseName。

如图 5.9 所示的 GradeBook 类现在包含了 5 个新增的 private 数据成员（第 20~24 行）——每个成绩级别（也就是 A、B、C、D 和 F）的计数器变量。它还包含了两个新增的 public 成员函数——inputGrades 和 displayGradeReport。成员函数 inputGrades 在第 16 行声明，它采用标记控制的循环从用户处读入任意多个用字母表示的成绩，并对每个输入的成绩更新相应的成绩计数器。而成员函数 displayGradeReport 的声明见第 17 行，它输出一份报告，其中包含了取得用字母表示的成绩的学生数。

源代码文件GradeBook.cpp（如图5.10所示）包含了GradeBook类的成员函数的定义。请注意构造函数中的第16~20行，它们分别将5个成绩计数器初始化为0，这意味着在一个GradeBook对象刚刚创建时，还没有成绩输入。可以看到，随着用户成绩的输入，这些计数器会在成员函数inputGrades中进行自增。成员函数setCourseName、getCourseName和displayMessage的定义仍然同于GradeBook类以前版本中的定义。下面让我们仔细地分析一下新的GradeBook成员函数。

```
1  // Fig. 5.10: GradeBook.cpp
2  // Member-function definitions for class GradeBook that
3  // uses a switch statement to count A, B, C, D and F grades.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include "GradeBook.h" // include definition of class GradeBook
10
11 // constructor initializes courseName with string supplied as argument;
12 // initializes counter data members to 0
13 GradeBook::GradeBook( string name )
14 {
15    setCourseName( name ); // validate and store courseName
16    aCount = 0; // initialize count of A grades to 0
17    bCount = 0; // initialize count of B grades to 0
18    cCount = 0; // initialize count of C grades to 0
19    dCount = 0; // initialize count of D grades to 0
20    fCount = 0; // initialize count of F grades to 0
21 } // end GradeBook constructor
22
23 // function to set the course name; limits name to 25 or fewer characters
24 void GradeBook::setCourseName( string name )
25 {
26    if ( name.length() <= 25 ) // if name has 25 or fewer characters
27       courseName = name; // store the course name in the object
28    else // if name is longer than 25 characters
29    { // set courseName to first 25 characters of parameter name
30       courseName = name.substr( 0, 25 ); // select first 25 characters
31       cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
32          << "Limiting courseName to first 25 characters.\n" << endl;
33    } // end if...else
34 } // end function setCourseName
35
36 // function to retrieve the course name
37 string GradeBook::getCourseName()
38 {
39    return courseName;
40 } // end function getCourseName
41
42 // display a welcome message to the GradeBook user
43 void GradeBook::displayMessage()
44 {
45    // this statement calls getCourseName to get the
46    // name of the course this GradeBook represents
47    cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
48       << endl;
49 } // end function displayMessage
50
51 // input arbitrary number of grades from user; update grade counter
52 void GradeBook::inputGrades()
53 {
54    int grade; // grade entered by user
```

图5.10　使用switch语句统计A、B、C、D和F级成绩人数的GradeBook类

```
55
56    cout << "Enter the letter grades." << endl
57       << "Enter the EOF character to end input." << endl;
58
59    // loop until user types end-of-file key sequence
60    while ( ( grade = cin.get() ) != EOF )
61    {
62       // determine which grade was entered
63       switch ( grade ) // switch statement nested in while
64       {
65          case 'A': // grade was uppercase A
66          case 'a': // or lowercase a
67             aCount++; // increment aCount
68             break; // necessary to exit switch
69
70          case 'B': // grade was uppercase B
71          case 'b': // or lowercase b
72             bCount++; // increment bCount
73             break; // exit switch
74
75          case 'C': // grade was uppercase C
76          case 'c': // or lowercase c
77             cCount++; // increment cCount
78             break; // exit switch
79
80          case 'D': // grade was uppercase D
81          case 'd': // or lowercase d
82             dCount++; // increment dCount
83             break; // exit switch
84
85          case 'F': // grade was uppercase F
86          case 'f': // or lowercase f
87             fCount++; // increment fCount
88             break; // exit switch
89
90          case '\n': // ignore newlines,
91          case '\t': // tabs,
92          case ' ': // and spaces in input
93             break; // exit switch
94
95          default: // catch all other characters
96             cout << "Incorrect letter grade entered."
97                << " Enter a new grade." << endl;
98             break; // optional; will exit switch anyway
99       } // end switch
100   } // end while
101 } // end function inputGrades
102
103 // display a report based on the grades entered by user
104 void GradeBook::displayGradeReport()
105 {
106    // output summary of results
107    cout << "\n\nNumber of students who received each letter grade:"
108       << "\nA: " << aCount // display number of A grades
109       << "\nB: " << bCount // display number of B grades
110       << "\nC: " << cCount // display number of C grades
111       << "\nD: " << dCount // display number of D grades
112       << "\nF: " << fCount // display number of F grades
113       << endl;
114 } // end function displayGradeReport
```

图 5.10　使用 switch 语句统计 A、B、C、D 和 F 级成绩人数的 GradeBook 类（续）



良好的编程习惯 5.11

在默认情况子句列在最后的switch语句中，该子句实际上不需要break语句。但是有些程序员出于程序清晰性以及与其他case子句相对称的目的，也会在其中使用break语句。

在输入中忽略换行符、制表符和空格

请注意图5.10的switch语句中的第90~93行，它们使程序跳过换行符、制表符和空格。一次读入一个字符会引起一些问题。为了让程序读入字符，我们必须通过按键盘上的回车键的方式，将它们送入计算机。这样，却会在希望处理的字符之后，在输入中多加了一个换行符。通常，这个换行符必须经过专门处理，才可以使程序正常工作。通过在switch语句中包含上述case子句，我们避免了在每次读入换行符、制表符和空格时，都由默认子句打印的一条错误信息。



常见的编程错误 5.10

在以一次一个的方式读入字符时，如果不对输入的换行符以及其他的空白间隔符进行处理，那么会导致逻辑错误。

测试 GradeBook 类

图5.11在第9行创建了一个GradeBook对象。第11行调用这个对象的成员函数displayMessage，向用户输出一条欢迎信息。第12行调用对象的成员函数inputGrades，从用户那里读入一组成绩，并记录获得每个级别成绩的学生人数。请注意，图5.11中的输入输出窗口显示了一条错误信息，它是对一个无效成绩（例如，E）输入的响应。第13行调用GradeBook成员函数displayGradeReport（定义在图5.10中的第104~114行），它输出了一份基于输入成绩的报告（如图5.11中输出所示）。

```
1  // Fig. 5.11: fig05_11.cpp
2  // Create GradeBook object, input grades and display grade report.
3
4  #include "GradeBook.h" // include definition of class GradeBook
5
6  int main()
7  {
8     // create GradeBook object
9     GradeBook myGradeBook( "CS101 C++ Programming" );
10
11    myGradeBook.displayMessage(); // display welcome message
12    myGradeBook.inputGrades(); // read grades from user
13    myGradeBook.displayGradeReport(); // display report based on grades
14    return 0; // indicate successful termination
15 } // end main
```

```
Welcome to the grade book for
CS101 C++ Programming!

Enter the letter grades.
Enter the EOF character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z

Number of students who received each letter grade:
A: 3
B: 2
C: 3
D: 2
F: 1
```

图 5.11　创建一个 GradeBook 对象并调用它的成员函数

switch 语句的 UML 活动图

图 5.12 显示了一般的 switch 多路选择语句的 UML 活动图。大多数的 switch 语句都在每个 case 中使用一个 break。这样，在处理当前 case 子句时，其 break 使 switch 语句结束。图 5.12 在活动图中通过包括 break 语句来强调这一点。假如没有包括 break 语句，那么处理完一条 case 后，控制不会转移到switch语句之后的第一条语句。取而代之的是，控制会转去处理下一个case语句中的动作。

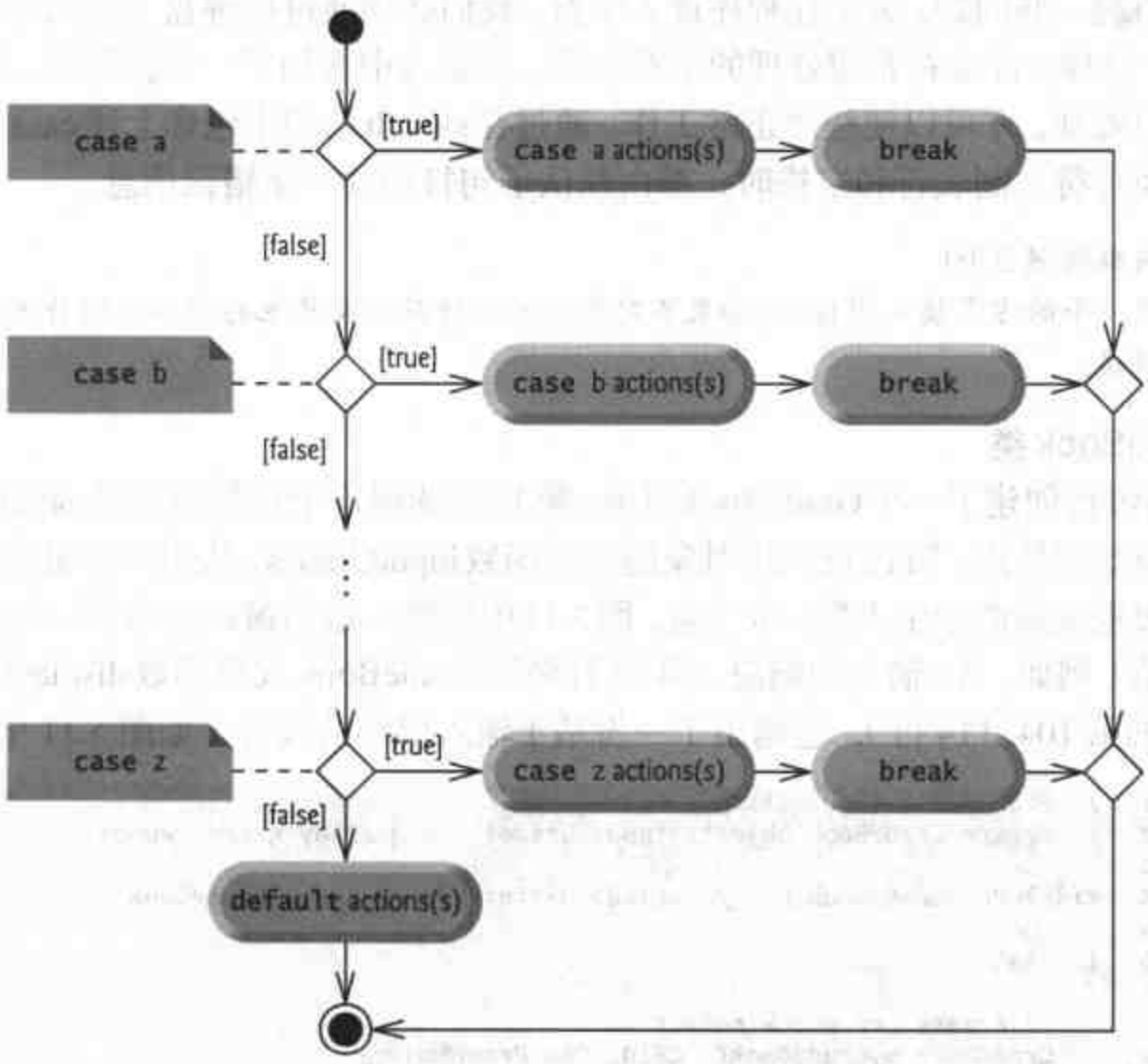


图 5.12　含有 break 语句的 switch 多路选择语句的 UML 活动图

这张图清楚地说明，在一个 case 子句末尾的 break 语句使程序控制马上退出 switch 语句。同样请注意，除一个初始状态、多个转换箭头、一个结束状态和几个注释外，此活动图包含了若干动作状态和判定符号。同时注意，该图中使用了合并符号，用于将来自 break 语句的转换合并到结束状态。

下面，请大家再想像一下：程序员有一个大柜子，其中包含了大量足够他使用的、空的 switch 语句的 UML 活动图。程序员的任务便是采用堆叠和嵌套的方式，将这些 switch 语句活动图同其他控制语句的活动图组合起来，以形成算法的结构化实现。然后，程序员再用适合于算法的动作表达式和监控条件填写动作状态和判定符号。请注意，虽然嵌套的控制语句很常见，但是在程序中很少发现嵌套的 switch 语句。

当使用 switch 语句时，请记住只可用它测试常整数表达式——字符常量和整数常量的任意组合，其计算结果是一个常整数值。一个字符常量表示为一对单引号括起来的特定字符，例如‘A’。一个整数常量就是一个整数值。此外，每个 case 标签只能指定一个常整数表达式。



常见的编程错误 5.11

在 switch 语句的 case 标签中指定包含变量的表达式（例如，a+b）是一个语法错误。

```
1  // Fig. 5.13: fig05_13.cpp
2  // break statement exiting a for statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int count; // control variable also used after loop terminates
10
11    for ( count = 1; count <= 10; count++ ) // loop 10 times
12    {
13       if ( count == 5 )
14          break; // break loop only if x is 5
15
16       cout << count << " ";
17    } // end for
18
19    cout << "\nBroke out of loop at count = " << count << endl;
20    return 0; // indicate successful termination
21 } // end main
```

```
1 2 3 4
Broke out of loop at count = 5
```

图 5.13　退出一条 for 循环语句的 break 语句

continue 语句

continue 语句在 while、for 或者 do...while 语句中执行时，使程序跳过循环体内剩下的语句，继续进行循环体的下一次迭代。在 while 和 do...while 语句中，循环继续条件的测试在 continue 语句执行之后马上进行。在 for 语句中，则执行增值表达式，然后对循环继续条件进行测试。

图 5.14 在 for 语句中使用 continue 语句（第 12 行），当嵌套的 if 语句（第 11~12 行）判定 count 的值为 5 时跳过输出语句（第 14 行）。当 continue 语句执行时，程序控制转到 for 头部（第 9 行）的控制变量增值部分继续执行，并且还会再循环 5 次。

```
1  // Fig. 5.14: fig05_14.cpp
2  // continue statement terminating an iteration of a for statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     for ( int count = 1; count <= 10; count++ ) // loop 10 times
10    {
11       if ( count == 5 ) // if count is 5,
12          continue;      // skip remaining code in loop
13
14       cout << count << " ";
15    } // end for
16
17    cout << "\nUsed continue to skip printing 5" << endl;
18    return 0; // indicate successful termination
19 } // end main
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

图 5.14　结束一条 for 语句单次迭代的 continue 语句

逻辑非（!）运算符

C++提供了!（逻辑非，也称为逻辑求反）运算符，使程序员可以“逆转”一个条件的含义。与组合了两个条件的&&和||二元运算符不一样，一元的逻辑非运算符只用一个条件作为操作数；一元的逻辑非运算符必须放在要“逆转”的条件的前面。当我们希望在原始条件（没有逻辑非运算符）为false时，选择一条感兴趣的执行路径，那么可采用逻辑非运算符。例如：

```
if ( !( grade == sentinelValue ) )
   cout << "The next grade is " << grade << endl;
```

其中括在grade == sentinelValue两边的圆括号是必需的，因为逻辑非运算符的优先级要比相等运算符的优先级高。

在大多数情况下，对于用逻辑非运算符表达的条件，程序员同样可以通过合适的关系运算符或者相等运算符来表达，从而避免使用逻辑非运算符。例如，前述的if语句也可以写成：

```
if ( grade != sentinelValue )
   cout << "The next grade is " << grade << endl;
```

这种灵活性常常可以帮助程序员以一种更加“自然”或者方便的形式表达条件。图5.17是逻辑非运算符（!）的真值表。

表达式	!表达式
false	true
true	false

图5.17　!（逻辑非）运算符的真值表

逻辑运算符的例子

图5.18通过产生相应的真值表演示逻辑运算符的使用情况。输出显示了所计算的每个表达式及其bool结果。在默认的情况下，bool值true和false由cout和流插入运算符分别显示为1和0。然而，我们在第11行使用流操纵元boolalpha指定每个bool表达式的值应该显示为单词“true”或者单词“false”的形式。例如，在第12行中表达式false && false的结果是false，所以输出的第2行包含的是单词“false”。第11~15行生成了&&运算符的真值表，第18~22行生成了||运算符的真值表，而第25~27行生成了!运算符的真值表。

```
1  // Fig. 5.18: fig05_18.cpp
2  // Logical operators.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::boolalpha; // causes bool values to print as "true" or "false"
7
8  int main()
9  {
10    // create truth table for && (logical AND) operator
11    cout << boolalpha << "Logical AND (&&)"
12       << "\nfalse && false: " << ( false && false )
13       << "\nfalse && true: " << ( false && true )
14       << "\ntrue && false: " << ( true && false )
15       << "\ntrue && true: " << ( true && true ) << "\n\n";
16
17    // create truth table for || (logical OR) operator
18    cout << "Logical OR (||)"
19       << "\nfalse || false: " << ( false || false )
20       << "\nfalse || true: " << ( false || true )
```

图5.18　逻辑运算符

```
      << "\ntrue || false: " << ( true || false )
      << "\ntrue || true: " << ( true || true ) << "\n\n";

   // create truth table for ! (logical negation) operator
   cout << "Logical NOT (!)"
      << "\n!false: " << ( !false )
      << "\n!true: " << ( !true ) << endl;
   return 0; // indicate successful termination
} // end main
```

```
Logical AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Logical OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Logical NOT (!)
!false: true
!true: false
```

图 5.18 逻辑运算符（续）

运算符的优先级和结合律的总结

图 5.19 将逻辑运算符添加到运算符的优先级和结合律表中。在这张表中，运算符从上到下按照优先级下降的顺序排列。

运算符						结合律	类型
()						从左向右	圆括号
++	--	static_cast< 类型 >()				从左向右	一元（后置）
++	--	+	-	!		从右向左	单目（前置）
*	/	%				从左向右	乘
+	-					从左向右	加
<<	>>					从左向右	插入 / 提取
<	<=	>	>=			从左向右	关系
==	!=					从左向右	相等
&&						从左向右	逻辑与
\|\|						从左向右	逻辑或
?:						从右向左	条件
=	+=	-=	*=	/=	%=	从右向左	赋值
,						从左向右	逗号

图 5.19 运算符的优先级和结合律

5.9 == 运算符与 = 运算符的混淆问题

无论经验如何，有一类错误是 C++ 程序员频繁要犯的错误。因此，我们认为非常有必要对此单列一节进行讨论。这个错误就是无意地互换了 ==（相等）和 =（赋值）运算符。换句话说，就是在该用 ==（相等）运算符的地方却写成了 =（赋值）运算符，或者在该用 =（赋值）运算符的地方却写成了 ==（相等）运算符。这种互换之所以具有破坏性，是因为它们通常不会产生语法错误。相反，含有这些错误语句的程序可以正常地通过编译并执行完毕。但是，由于隐含了运行时的逻辑错

前面曾经介绍过，结构化编程产生的程序比非结构化的程序更容易理解、测试、调试和修改，甚至在数学意义上证明了结构化编程是正确的。

图 5.20 利用活动图总结了 C++ 的控制语句，其中的初始状态和结束状态表示每条语句是单入口和单出口的。如果任意连接在活动图中的各个符号，可能导致非结构化的程序。因此，编程专业人员只使用有限的一组控制语句，通过仅有的两种简单方式将它们组合起来，构造出结构化的程序。

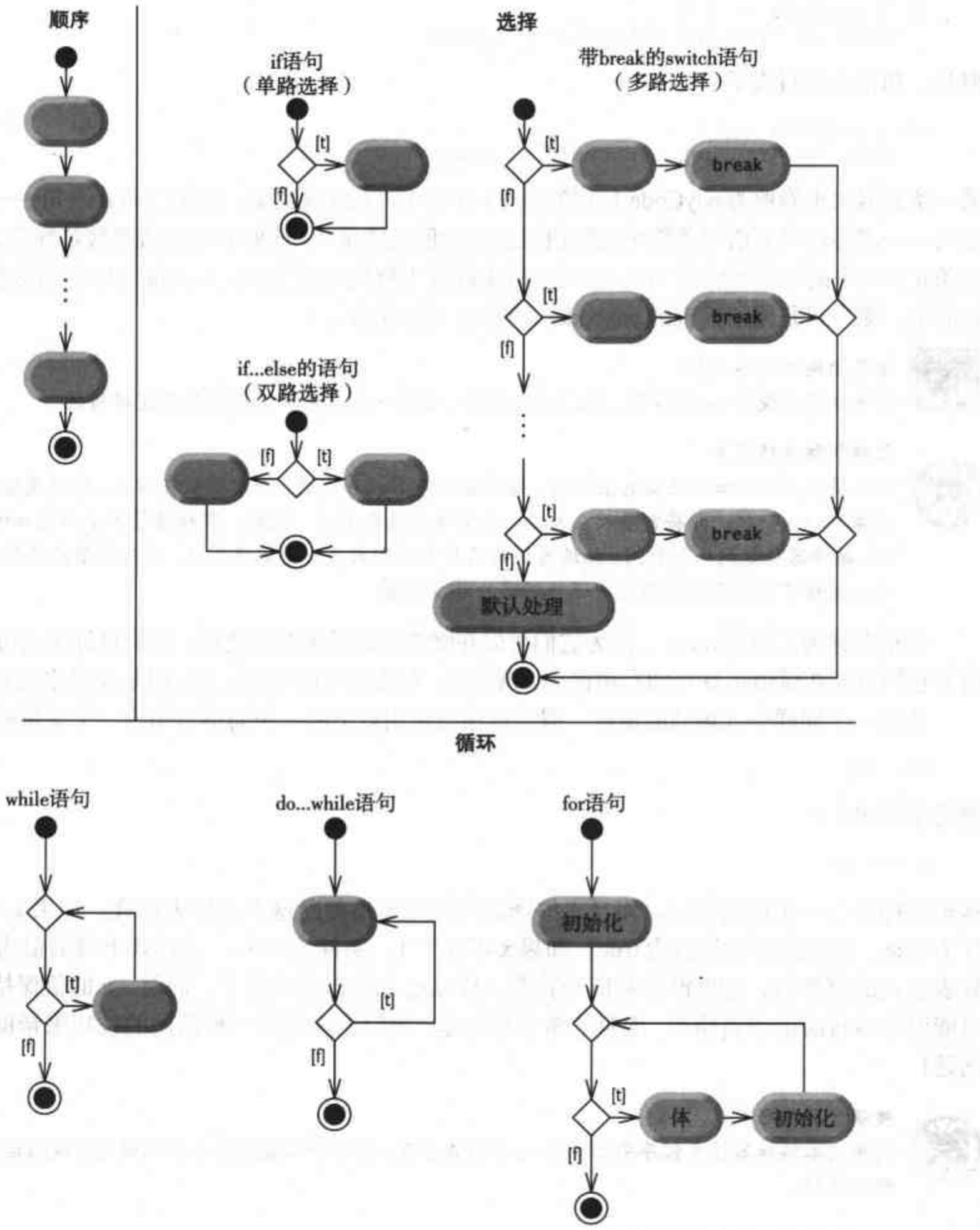


图 5.20 C++ 的单入口 / 单出口的顺序、选择和循环语句

为了简单起见，仅使用单入口 / 单出口的控制语句——每条控制语句只有一条道进入，并且只有一条道退出。按顺序连接控制语句构成结构化程序是很简单的—— 一条控制语句的结束状态连

接到下一条控制语句的初始状态——就是说，控制语句在程序中一个接一个地放置。我们称这种方式为“控制结构的堆叠”。构成结构化程序的规则还允许控制语句进行嵌套。

图 5.21 列出了正确构建结构化程序的规则。这些规则假定可以用动作状态象征任何动作，而且假定从图5.22所示的最简单的活动图开始。这个所谓的最简单的活动图仅由一个初始状态、一个动作状态、一个结束状态和若干个转换箭头组成。

形成结构化程序的规则

1) 从图 5.22 所示的“最简单的活动图”开始；
2) 任何一个动作状态都可以被两个顺次的动作状态所取代；
3) 任何一个动作状态都可以被任何控制语句（顺序语句、if 语句、if...else 语句、switch 语句、while 语句、do... while 语句或者 for 语句）所取代；
4) 规则 2 和规则 3 可根据自己的需要以任何顺序随意应用。

图 5.21　形成结构化程序的规则

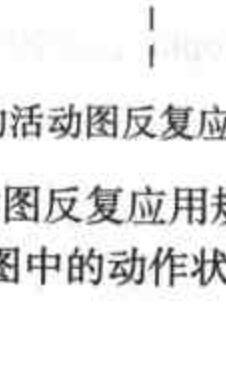


图 5.22　最简单的活动图

应用图 5.21 所示规则总是可以得到一张整洁的、构建块状的活动图。例如，对最简单的活动图反复应用规则 2，如图 5.23 所示，可以得到一张包含了许多顺序放置的动作状态的活动图。规则 2 产生一系列控制语句的堆栈，所以称规则 2 为堆叠规则。注意：图 5.23 中的垂直虚线并不是 UML 的一部分，我们用它们来划分演示图 5.21 规则 2 应用的 4 张活动图。

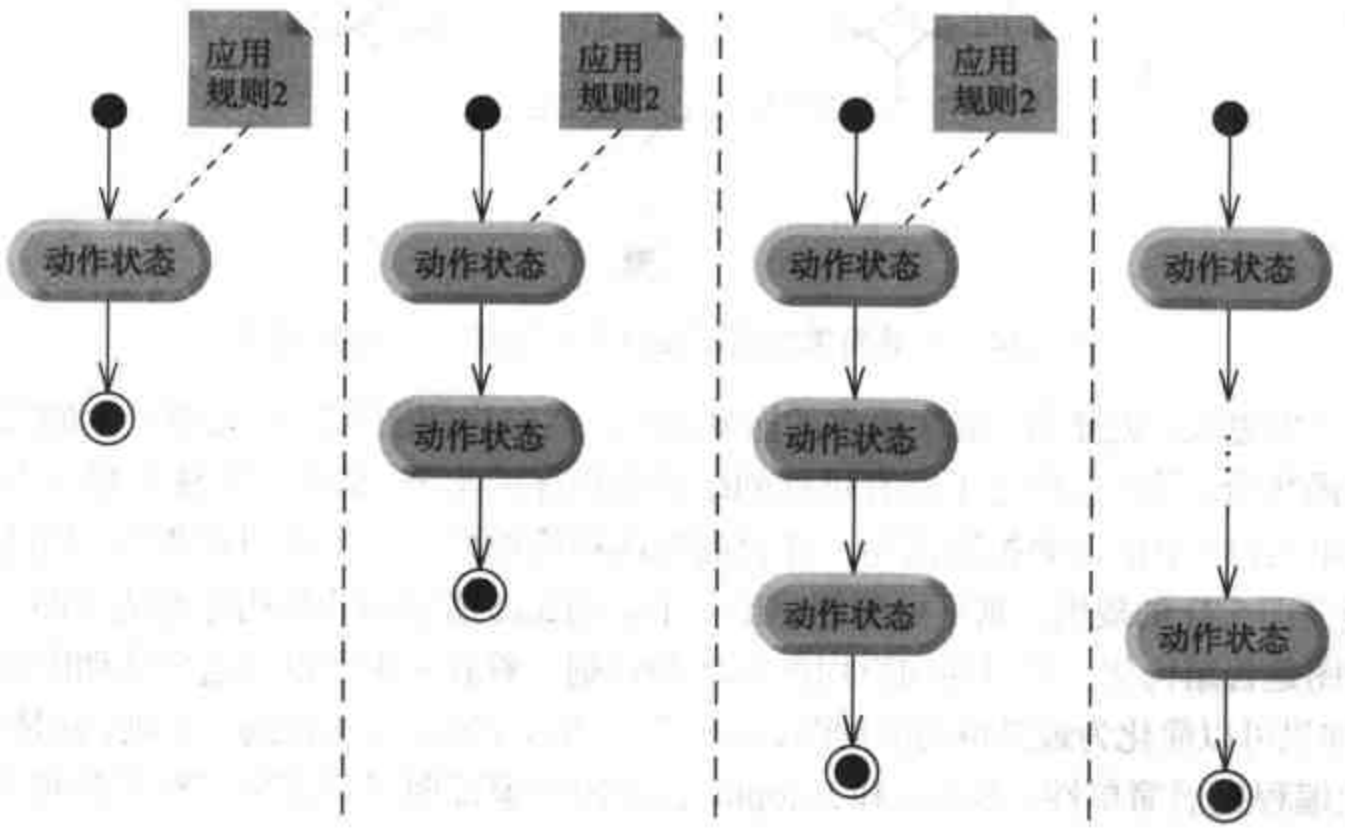


图 5.23　对最简单的活动图反复应用图 5.21 的规则 2

规则 3 称为嵌套规则。对最简单的活动图反复应用规则 3，可以得到一张控制语句巧妙嵌套的活动图。例如，在图 5.24 中，最简单的活动图中的动作状态被一条双路选择（if...else）语句所代替。

然后，对此双路选择语句中的动作状态再次应用规则3，即各用一条双路选择语句分别取代其中的每个动作状态。围绕每个双路选择语句的虚线状态符号表示现已取代的前一张活动图中的动作状态。注意：图 5.24 显示的虚线箭头和虚线状态符号都不是 UML 的成分。在此用它们是出于教学目的，以便阐明任何动作状态都可以用一条控制语句来替换。

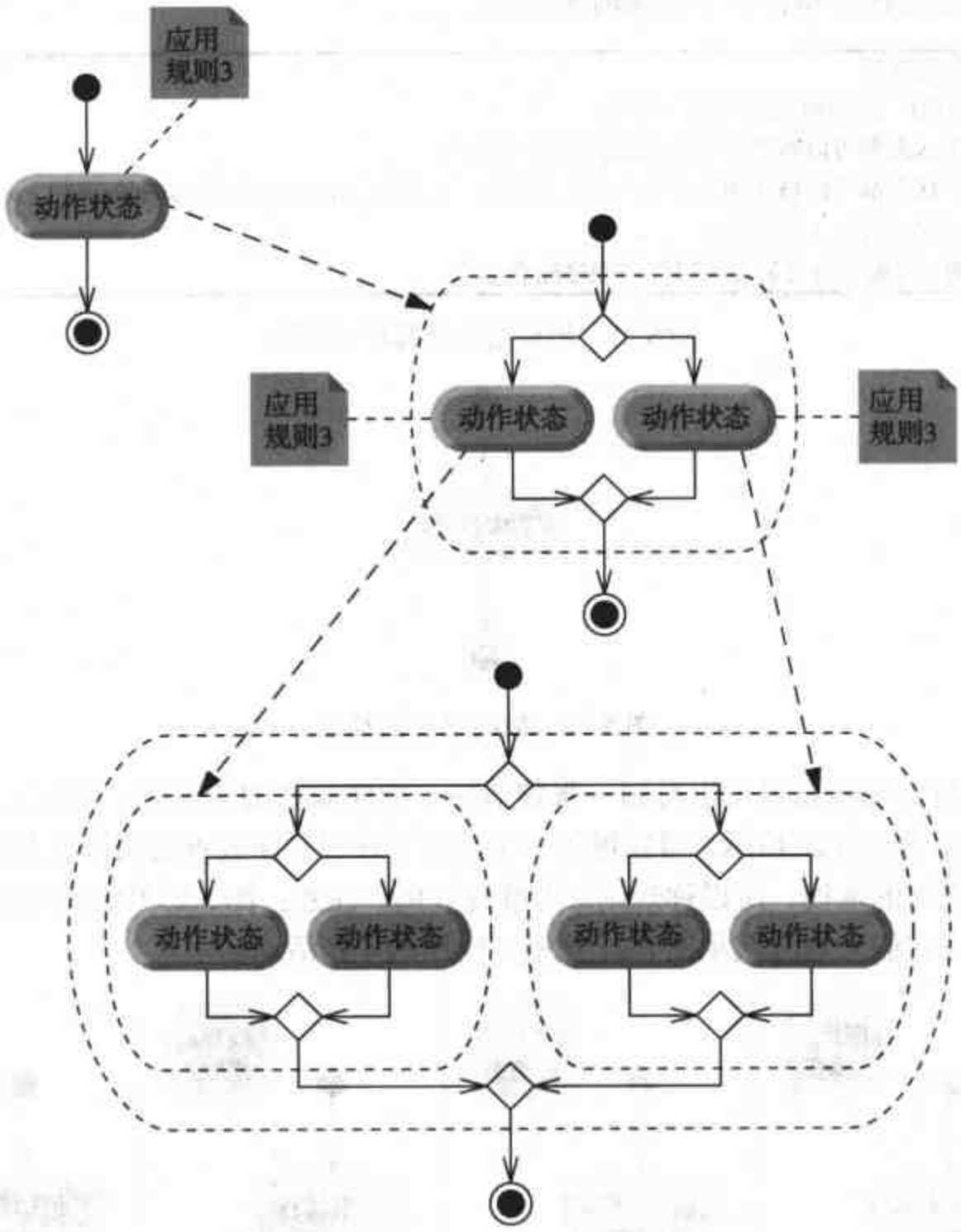


图 5.24　对最简单的活动图反复应用图 5.21 的规则 3

规则4产生更大、更复杂、层次更多的嵌套语句。通过应用图5.21的规则产生的图组成了所有可能的活动图集合，因此也产生了所有可能的结构化程序的集合。结构化方法的魅力在于我们只需要使用7种单入口/单出口的控制语句，并只需要两种简单的方式，就可以把它们组装起来。

如果遵循图5.21的规则，则不可能创建出一个如图5.25所示的非结构化的活动图。如果不能确定某个活动图是否结构化，可以逆向应用图 5.21 的规则，看看是否可以将这个活动图简化到最简单的活动图。如果可以简化为最简单的活动图，那么原始的活动图是结构化的；否则，就是非结构化的。

结构化编程提倡简单性。Böhm 和 Jacopini 已经向大家证明了只需要三种控制形式：

- 顺序
- 选择
- 循环

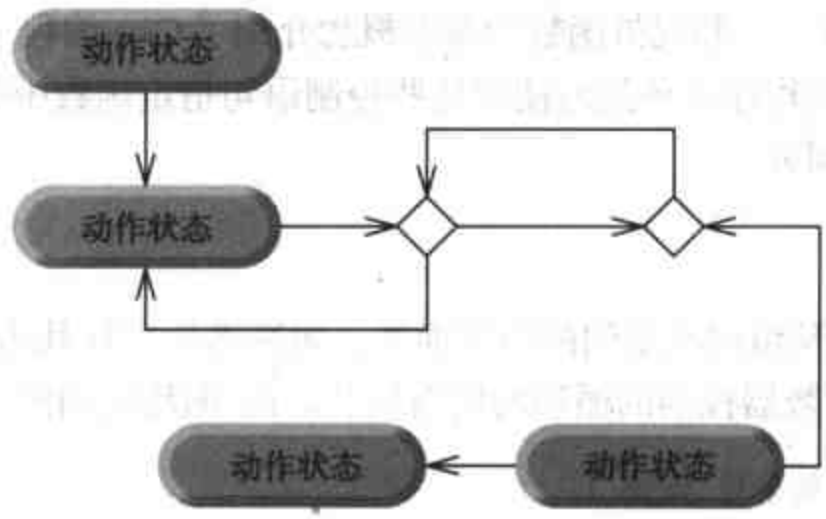


图 5.25　非结构化的活动图

其中，顺序结构是最平常的结构。只是按照语句应该执行的顺序列出它们而已。

选择可以用以下三种方法中的一种实现：

- if 语句（单路选择）
- if...else 语句（双路选择）
- switch 语句（多路选择）

事实上很容易证明简单的 if 语句即可提供任何形式的选择——任何能用 if...else 语句和 switch 语句完成的工作，也可以通过简单 if 语句的组合来实现（尽管有可能不够清晰和高效）。

循环可以用以下三种方法中的一种实现：

- while 语句
- do...while 语句
- for 语句

同样很容易证明，while语句亦足以实现任何形式的循环。也就是说，任何能用do...while和for语句做到的事情，都可以用 while 语句来实现（但是有可能不够流畅）。

综上所述，一个 C++ 程序中所需的任何控制形式均可以用下列形式表示:

- 顺序
- if 语句（选择）
- while 语句（循环）

并且这些控制语句只需要通过两种方式进行组合，即堆叠方式和嵌套方式。毫无疑问，结构化编程提倡简单性。

5.11　本章小结

在这一章中，我们完成了对 C++ 控制语句的介绍。现在，程序员可以利用它们控制函数中的执行流程。第4章讨论了 if、if...else 和 while 语句，而本章讨论了余下的C++控制语句——for、do...while 和 switch 语句。我们已说明用顺序结构（即，按照应该执行的顺序列出的语句）、三种选择语句（if、if...else 和 switch）和三种循环语句（for、do...while 和 while）可以开发出任何算法。本章和第4章讨论了程序员在组合这些构建块的过程中如何利用已证明的程序构建和解决问题的方法。本章还介绍了C++的逻辑运算符，使程序员可以在控制语句中使用更复杂的条件表达式。最后，我们分析了混淆相等和赋值运算符这一常见错误，并提供了避免此类错误的建议。

a)
```
For ( x = 100, x >= 1 , x++ )
   cout << x << endl;
```

b) 下面的代码判断整数 value 是奇数还是偶数，并打印结果。

```
switch ( value % 2 )
{
   case 0:
      cout << "Even integer" << endl;
   case 1:
      cout << "Odd integer" << endl;
}
```

c) 下面的代码输出从 19 开始到 1 为止的所有奇数。

```
for ( x = 19; x >= 1 ; x += 2 )
   cout << x << endl;
```

d) 下面的代码输出 2 到 100 的所有偶数。

```
counter = 2 ;

do
{
   cout << counter << endl;
   counter += 2 ;
} While ( counter < 100 );
```

5.5 编写一个程序，使用一条 for 语句求一系列整数之和。假定输入的第一个整数指定了继续输入的值的个数。要求程序的每条输入语句只读入一个值。一次运行的示范输入序列如下：

```
5 100 200 300 400 500
```

其中的 5 表明后续的 5 个输入是要求求和的值。

5.6 编写一个程序，使用一条for语句求若干个整数的平均值并打印结果。假定最后读入的值是标记值 9999。一次运行的示范输入序列如下：

```
10 8 11 7 9 9999
```

表明程序应计算 9999 之前的所有值的平均值。

5.7 请写出下面程序的输出结果：

```
1  // Exercise 5.7: ex05_07.cpp
2  // What does this program print?
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  int main()
9  {
10    int x; // declare x
11    int y; // declare y
12
13    // prompt user for input
14    cout << "Enter two integers in the range 1-20: ";
```

```
   cin >> x >> y;  // read values for x and y
   for ( int i = 1; i <= y; i++ ) // count from 1 to y
   {
      for ( int j = 1; j <= x; j++ ) // count from 1 to x
         cout << '@'; // output @
      cout << endl; // begin new line
   } // end outer for
   return 0; // indicate successful termination
} // end main
```

5.8 编写一个程序，使用一条for语句找到一组整数中的最小整数。假定输入的第一个值指定了继续输入的值的个数，它并不是参与比较的一个整数。

5.9 编写一个程序，使用一条for语句计算并打印从1到15中奇数的乘积。

5.10 在概率问题中常常用到阶乘函数。请利用习题4.35定义的阶乘编写一个程序。该程序使用一条for语句求1到5每个整数的阶乘，要求以表格的形式打印出结果。请问如果计算20的阶乘，可能会遇到什么麻烦？

5.11 修改5.4节的复利程序，可以重复计算利率为5%、6%、7%、8%、9%和10%时的复利金额。请使用一条for语句改变利率。

5.12 编写一个程序，使用for语句分别打印以下图案，要求下一个图案在上一个图案的下方。利用for循环生成这些图案。所有的星号（*）都用单条的语句cout << '*'; 来打印，这使星号可以挨在一起。提示：最后的两个图案需要每行以适当数量的空格开头。选做题：巧妙地使用嵌套的for循环，把用于解决这4个问题的代码组合到一个程序中，并排地打印这4个图案。

(a)
```
*
**
***
****
*****
******
*******
********
*********
**********
```

(b)
```
**********
*********
********
*******
******
*****
****
***
**
*
```

(c)
```
**********
 *********
  ********
   *******
    ******
     *****
      ****
       ***
        **
         *
```

(d)
```
         *
        **
       ***
      ****
     *****
    ******
   *******
  ********
 *********
**********
```

5.13 计算机的一个有趣的应用是绘制图表和直方图。请编写一个程序，要求用户输入5个数（都在1到30之间）。假定用户只输入有效的值。对于用户输入的每个数，程序都打印一行紧挨在一起的星号，星号的个数等于输入的数。例如，如果程序读入数7，那么它打印*******。

5.14 一家邮购公司销售5种不同的产品，零售价分别是：产品1，$2.98；产品2，$4.50；产品3，$9.98；产品4，$4.49和产品5，$6.87。请编写一个程序，要求用户输入一系列如下所示的数：

a) 产品编号

b) 销售量

5.22（摩根定律）在本章中，我们讨论了逻辑运算符 &&、|| 和 !。利用摩根定律，有时可以更方便地表示一个逻辑表达式。该定律指出，表达式 !（条件 1 && 条件 2）在逻辑上等价于表达式（!条件 1 || !条件 2）。同样，表达式 !（条件 1 || 条件 2）在逻辑上等价于表达式（!条件 1 && ! 条件 2）。请根据摩根定律，书写出与下述表达式等价的表达式。然后编写程序证明每个原始的表达式和对应的新表达式确实是等价的。

a) `!( x < 5 ) && !( y >= 7 )`

b) `!( a == b ) || !( g != 5 )`

c) `!( ( x <= 8 ) && ( y > 4 ) )`

d) `!( ( i > 4 ) || ( j <= 6 ) )`

5.23 编写一个程序，打印以下的菱形。要求输出的语句要么打印一个星号（*），要么打印一个空格。尽量多用循环（使用嵌套 for 语句），同时尽量减少输出语句的使用次数。

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

5.24 修改习题 5.23 所编写的程序，要求读入 1 到 19 范围内的一个奇数，指定菱形中的行数，然后显示适合此尺寸的一个菱形。

5.25 break 和 continue 语句遭到质疑的原因是它们的非结构化性。实际上，break 和 continue 语句总能用结构化的语句取而代之，尽管实现起来不怎么方便。请详述如何从程序的一条循环语句中去除 break 语句，并用某种结构化的手段替代。提示：break 语句用于在循环体内离开一个循环。另一个离开的办法是让循环继续条件测试失败。请考虑在循环继续条件测试中利用另一个测试，指出“由于符合一个‘break’条件，所以提前退出”。请使用在此介绍的方法替换图 5.13 中的 break 语句。

5.26 请指出以下程序段的用途？

```
1  for ( int i = 1; i <= 5; i++ )
2  {
3     for ( int j = 1; j <= 3; j++ )
4     {
5        for ( int k = 1; k <= 4; k++ )
6           cout << '*';
7
8        cout << endl;
9     } // end inner for
10
11    cout << endl;
12 } // end outer for
```

5.27 详述如何从程序的一条循环语句中去除 continue 语句，并用某种结构化的手段替代。请使用在此介绍的方法替换图 5.14 中的 continue 语句。

C++标准库提供了丰富的函数，用于执行通用数学计算、字符串操作、字符处理、输入/输出、错误检查以及其他许多有用操作。这样，程序员的工作就简单了许多，因为这些函数很大程度上满足了程序员的需求。C++ 标准库函数是 C++ 编程环境不可分割的一部分。



软件工程知识 6.1
阅读当前所使用的编译器的相关文档，熟悉 C++ 标准库中的相关函数和类。

函数（在其他编程语言中也称为方法或者过程）允许程序员通过将任务分解成自我包含的单元实现程序的模块化。读者在编写的每个程序中都用到了函数。这些函数往往称为用户自定义函数（user-defined function）或者程序员自定义函数（programmer-defined function）。函数体中的语句只需要书写一次，就能够在程序的多个地方重用，并且对于其他函数来说函数体的内容是不可见的。

用函数模块化程序有以下几个动机：一个是分而治之法的使用，这种方法通过由小的、简单的部件构建程序，使得程序的开发更易于控制；另外一个是软件重用性，以现有的函数作为构建块来创建新的程序。例如，在前面的程序中，我们不必定义如何从键盘读入一行文本，因为 C++ 通过 <string>头文件中的 getline 函数提供了这个功能；第三个动机是避免重复代码。此外，把程序分成有意义的函数也使其更容易调试和维护。



软件工程知识 6.2
为了提高软件的可重用性，每个函数应该限定于单一的、良好定义的任务，并且函数名应该有效地表达任务。这样的函数才能使程序更易于编写、测试、调试和维护。



错误预防技巧 6.1
一个只执行单个任务的小函数的测试和调试要比一个执行很多任务的大函数的测试和调试容易得多。



软件工程知识 6.3
如果不能为函数起一个能表达其任务的简洁的名字，那么该函数很可能试图执行太多的不同任务。通常，最好把这样的函数拆分成多个小函数。

我们知道，函数是通过函数调用而行动的。当被调用的函数完成它的任务后，它要么返回结果，要么简单地把控制权还给调用者。这种程序结构类似于公司里的分级管理层次（见图 6.1）。老板（boss）（与调用函数相类似）要求员工（worker）（与被调用函数相类似）完成一项任务并且汇报（也就是返回）完成任务后的结果。boss 函数并不知道 worker 函数是如何完成其指定的任务的。worker 函数也可能调用其他的 worker 函数，boss 函数也是不知道的。这种实现细节的隐藏促进了良好的软件工程。图 6.1 所示的是 boss 函数以层次的方式同多个 worker 函数进行通信。boss 函数把任务分给不同的 worker 函数来完成。请注意，worker1 也是 worker4 和 worker5 的“boss 函数”。

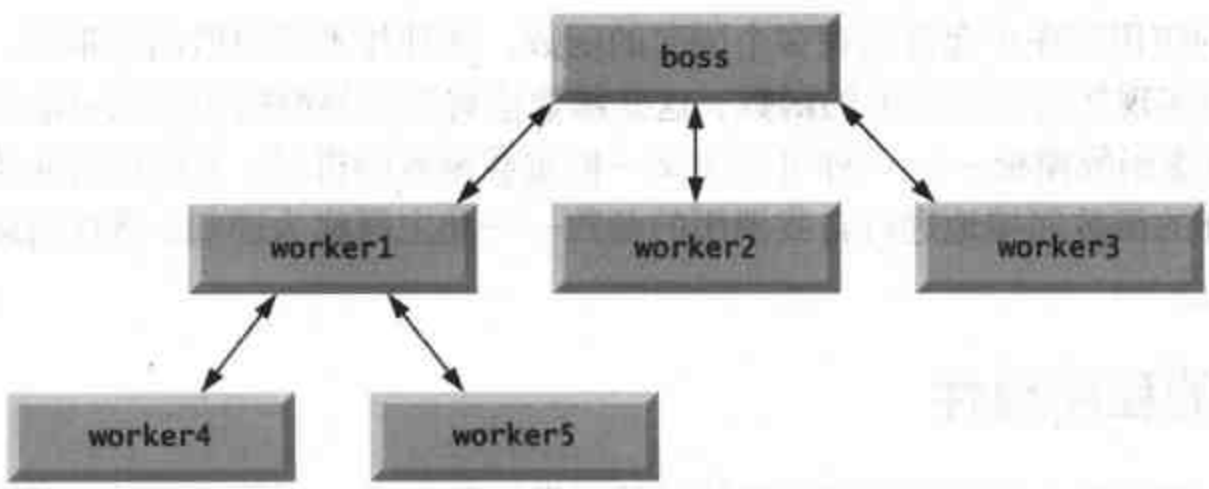


图 6.1　层次结构的 boss 函数 /worker 函数关系

6.4 多参数函数定义

在第3章至第5章中，我们讲述的类包含的简单函数至多只有一个参数。函数常常需要多条信息来执行它的任务。因此我们现在讨论具有多个参数的函数。

图6.3至图6.5中的程序对GradeBook类进行了修改，添加了一个用户自定义函数maximum，该函数可以在3个int类型的值中确定并且返回最大值。当这个应用程序开始执行时，main函数（见图6.5中的第5~14行）创建GradeBook类的一个对象（第8行），并调用该对象的inputGrades成员函数（第11行）从用户处读取3个整型成绩。在GradeBook类的实现文件中（见图6.4），成员函数inputGrades中的第54～55行提示用户输入3个整型值并读入用户输入。第58行调用成员函数maximum（定义在第62～75行），该函数确定最大值，然后return语句（第74行）在inputGrades函数调用maximum函数处（第58行）返回该值。随后成员函数inputGrades将此返回值存储到数据成员maximumGrade中。这个值然后通过调用displayGradeReport函数来输出（见图6.5的第12行）。注意：我们之所以把这个函数命名为displayGradeReport，是因为GradeBook类的后续版本中会使用该函数显示一张全面的成绩报告，包括最高分和最低分。在第7章中，我们会进一步增强GradeBook类，使其能够处理任意数量的成绩。

```
1  // Fig. 6.3: GradeBook.h
2  // Definition of class GradeBook that finds the maximum of three grades.
3  // Member functions are defined in GradeBook.cpp
4  #include <string> // program uses C++ standard string class
5  using std::string;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    GradeBook( string ); // constructor initializes course name
12    void setCourseName( string ); // function to set the course name
13    string getCourseName(); // function to retrieve the course name
14    void displayMessage(); // display a welcome message
15    void inputGrades(); // input three grades from user
16    void displayGradeReport(); // display a report based on the grades
17    int maximum( int, int, int ); // determine max of 3 values
18 private:
19    string courseName; // course name for this GradeBook
20    int maximumGrade; // maximum of three grades
21 }; // end class GradeBook
```

图6.3 GradeBook头文件

```
1  // Fig. 6.4: GradeBook.cpp
2  // Member-function definitions for class GradeBook that
3  // determines the maximum of three grades.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include "GradeBook.h" // include definition of class GradeBook
10
11 // constructor initializes courseName with string supplied as argument;
12 // initializes studentMaximum to 0
13 GradeBook::GradeBook( string name )
14 {
15    setCourseName( name ); // validate and store courseName
16    maximumGrade = 0; // this value will be replaced by the maximum grade
17 } // end GradeBook constructor
```

图6.4 GradeBook类定义函数maximum

```
18
19 // function to set the course name; limits name to 25 or fewer characters
20 void GradeBook::setCourseName( string name )
21 {
22    if ( name.length() <= 25 ) // if name has 25 or fewer characters
23       courseName = name; // store the course name in the object
24    else // if name is longer than 25 characters
25    { // set courseName to first 25 characters of parameter name
26       courseName = name.substr( 0, 25 ); // select first 25 characters
27       cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
28          << "Limiting courseName to first 25 characters.\n" << endl;
29    } // end if...else
30 } // end function setCourseName
31
32 // function to retrieve the course name
33 string GradeBook::getCourseName()
34 {
35    return courseName;
36 } // end function getCourseName
37
38 // display a welcome message to the GradeBook user
39 void GradeBook::displayMessage()
40 {
41    // this statement calls getCourseName to get the
42    // name of the course this GradeBook represents
43    cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
44       << endl;
45 } // end function displayMessage
46
47 // input three grades from user; determine maximum
48 void GradeBook::inputGrades()
49 {
50    int grade1; // first grade entered by user
51    int grade2; // second grade entered by user
52    int grade3; // third grade entered by user
53
54    cout << "Enter three integer grades: ";
55    cin >> grade1 >> grade2 >> grade3;
56
57    // store maximum in member studentMaximum
58    maximumGrade = maximum( grade1, grade2, grade3 );
59 } // end function inputGrades
60
61 // returns the maximum of its three integer parameters
62 int GradeBook::maximum( int x, int y, int z )
63 {
64    int maximumValue = x; // assume x is the largest to start
65
66    // determine whether y is greater than maximumValue
67    if ( y > maximumValue )
68       maximumValue = y; // make y the new maximumValue
69
70    // determine whether z is greater than maximumValue
71    if ( z > maximumValue )
72       maximumValue = z; // make z the new maximumValue
73
74    return maximumValue;
75 } // end function maximum
76
77 // display a report based on the grades entered by user
78 void GradeBook::displayGradeReport()
79 {
80    // output maximum of grades entered
81    cout << "Maximum of grades entered: " << maximumGrade << endl;
82 } // end function displayGradeReport
```

图 6.4 GradeBook 类定义函数 maximum（续）

```
1  // Fig. 6.5: fig06_05.cpp
2  // Create GradeBook object, input grades and display grade report.
3  #include "GradeBook.h" // include definition of class GradeBook
4
5  int main()
6  {
7     // create GradeBook object
8     GradeBook myGradeBook( "CS101 C++ Programming" );
9
10    myGradeBook.displayMessage(); // display welcome message
11    myGradeBook.inputGrades(); // read grades from user
12    myGradeBook.displayGradeReport(); // display report based on grades
13    return 0; // indicate successful termination
14 } // end main
```

```
Welcome to the grade book for
CS101 C++ Programming!

Enter three integer grades: 86 67 75
Maximum of grades entered: 86
```

```
Welcome to the grade book for
CS101 C++ Programming!

Enter three integer grades: 67 86 75
Maximum of grades entered: 86
```

```
Welcome to the grade book for
CS101 C++ Programming!

Enter three integer grades: 67 75 86
Maximum of grades entered: 86
```

图6.5 演示maximum函数



软件工程知识6.4

图6.4中第58行的用于分隔maximum函数实参的逗号并不是5.3节中讨论的逗号运算符。逗号运算符保证它的操作数从左到右进行求值。然而，函数实参的求值顺序不是由C++标准来指定的。因此，不同的编译器就会有不同的求函数实参值的顺序。C++标准的确保证在被调用函数执行之前，函数调用中的所有实参都要求值。



可移植性提示6.1

有时候，函数的实参是较复杂的表达式，其中又调用其他的函数。在这样的情况下，编译器对函数实参的求值顺序会影响若干实参的值。如果编译器之间的求值顺序不同，那么传递给函数的实参值也可能会不同，从而导致难以捉摸的逻辑错误。



错误预防技巧6.2

如果用户对函数实参的求值顺序以及是否由此影响传递给函数的值有所疑惑的话，那么在调用函数之前用单独的赋值语句对实参进行求值，把每个表达式的结果赋给局部变量，然后再将这些变量作为函数的实参。

成员函数maximum的原型（见图6.3中的第17行）表明该函数的返回值是整型数据、函数名为maximum并且需要三个整型参数来完成它的任务。函数maximum的头（见图6.4中的第62行）与函数原型相匹配，指明了形参名是x，y和z。当调用maximum时（见图6.4中的第58行），参数x

间的数字。而在某视频游戏中随机预测穿过地平线的下一艘飞船的类型的程序可能需要的是 1 到 4 之间的数字。

投掷六面骰子

为了说明 rand 函数，我们开发了一个程序（见图 6.8）来模拟投掷六面骰子 20 次，并且打印出每次的结果。rand 函数的原型在<cstdlib>中。为了生成 0 到 5 之间的整型值，我们使用 rand 函数和取模运算符（%）如下：

```
rand() % 6
```

这种方法称为比例缩放（scaling）。数字 6 称为比例缩放因子（scaling factor）。然后，我们可以对取模操作的结果加 1 来变换生成数字的范围。图 6.8 证实了结果确实是在 1 到 6 之间。

```
1  // Fig. 6.8: fig06_08.cpp
2  // Shifted and scaled random integers.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 #include <cstdlib> // contains function prototype for rand
11 using std::rand;
12
13 int main()
14 {
15    // loop 20 times
16    for ( int counter = 1; counter <= 20; counter++ )
17    {
18       // pick random number from 1 to 6 and output it
19       cout << setw( 10 ) << ( 1 + rand() % 6 );
20
21       // if counter is divisible by 5, start a new line of output
22       if ( counter % 5 == 0 )
23          cout << endl;
24    } // end for
25
26    return 0; // indicates successful termination
27 } // end main
```

```
     6         6         5         5         6
     5         1         1         5         3
     6         6         2         4         2
     6         2         3         4         1
```

图 6.8　由 1 + rand()% 6 生成的经偏移、比例缩放后的整数

投掷六面骰子 6 000 000 次

为了表明函数 rand 生成其区间内每个数字的机率近乎相等，图 6.9 模拟了 6 000 000 次投掷六面骰子的情况。1 到 6 之间的每个整数出现的次数都应该接近 1 000 000 次。图 6.9 结尾处的输出窗口证实了这一结果。

正如程序的输出所示，我们可以通过对 rand 函数生成的数值进行比例缩放和偏移来模拟六面骰子的投掷。请注意程序应当永远不会到达 switch 结构的 default 情况（第 50~51 行），因为 switch 的控制表达式（face）总是具有 1 ~ 6 范围中的值。可是我们仍提供了 default 情况，这是一种良好的编程习惯。在学习了第 7 章之后，我们会给出如何用单行语句优雅地替换图 6.9 中的整个 switch 结构。

```
// Fig. 6.9: fig06_09.cpp
// Roll a six-sided die 6,000,000 times.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib> // contains function prototype for rand
using std::rand;

int main()
{
   int frequency1 = 0; // count of 1s rolled
   int frequency2 = 0; // count of 2s rolled
   int frequency3 = 0; // count of 3s rolled
   int frequency4 = 0; // count of 4s rolled
   int frequency5 = 0; // count of 5s rolled
   int frequency6 = 0; // count of 6s rolled

   int face; // stores most recently rolled value

   // summarize results of 6,000,000 rolls of a die
   for ( int roll = 1; roll <= 6000000; roll++ )
   {
      face = 1 + rand() % 6; // random number from 1 to 6

      // determine roll value 1-6 and increment appropriate counter
      switch ( face )
      {
         case 1:
            ++frequency1; // increment the 1s counter
            break;
         case 2:
            ++frequency2; // increment the 2s counter
            break;
         case 3:
            ++frequency3; // increment the 3s counter
            break;
         case 4:
            ++frequency4; // increment the 4s counter
            break;
         case 5:
            ++frequency5; // increment the 5s counter
            break;
         case 6:
            ++frequency6; // increment the 6s counter
            break;
         default: // invalid value
            cout << "Program should never get here!";
      } // end switch
   } // end for

   cout << "Face" << setw( 13 ) << "Frequency" << endl; // output headers
   cout << "   1" << setw( 13 ) << frequency1
      << "\n   2" << setw( 13 ) << frequency2
      << "\n   3" << setw( 13 ) << frequency3
      << "\n   4" << setw( 13 ) << frequency4
      << "\n   5" << setw( 13 ) << frequency5
      << "\n   6" << setw( 13 ) << frequency6 << endl;
   return 0; // indicates successful termination
} // end main
```

```
Face    Frequency
   1       999702
   2      1000823
   3       999378
   4       998898
   5      1000777
   6      1000422
```

图 6.9　投掷六面骰子 6 000 000 次



错误预防技巧 6.3

即使绝对确信没有 bug，也应该在 switch 中提供 default 的情况来处理错误。

随机数生成器的随机化

再次执行图 6.8 中的程序产生如下输出：

```
6       6       5       5       6
5       1       1       5       3
6       6       2       4       2
6       2       3       4       1
```

请注意，程序输出的序列同图 6.8 中的完全一样。这些序列怎么可能是随机的数字呢？具有讽刺意味的是，这种重复性正是函数 rand 的一个重要特性。在调试模拟程序时，这种重复性是证明程序能正常运行的一个关键。

rand 函数实际上生成的是伪随机数。重复地调用 rand 会生成看似随机的数字序列。然而，程序每次执行时产生的序列都是重复的。一旦程序彻底调试完毕，就可以将它调整为在每次执行时都产生不同的随机数序列的程序。这个过程称为随机化（randomizing），可以通过 C++ 标准库函数 srand 来实现。srand 函数接收一个 unsigned 整型参数，作为函数 rand 的种子，用来在每次程序执行时都生成不同的随机数序列。

图 6.10 是使用 srand 函数的程序。程序中的 unsigned 数据类型是 unsigned int 的缩写方式。int 类型的值至少需要两个字节的内存来保存（在现在流行的 32 位系统上常常使用 4 字节内存），并且可以是正值或者负值。unsigned int 类型的变量也至少需要两个字节的内存来保存。两字节的 unsigned int 只能取 0 ~ 65535 范围内的非负值。四字节的 unsigned int 只能取 0 ~ 4294967295 之间的非负值。srand 函数以一个 unsigned int 值作为参数，其函数原型在<cstdlib>头文件中。

我们多运行几次程序来观察结果。请注意，假如用户输入的是不同的种子，程序每次执行都会产生不同的随机数字序列。我们在第 1 个和第 3 个输出示范中使用了相同的种子，结果它们产生了相同的 10 个数字的序列。

为了在随机化时不用每次都输入种子，可以使用如下语句：

```
srand( time( 0 ) );
```

这条语句使计算机通过读取自己的时钟来获得种子值。time 函数（在前面的语句中其实参为 0）返回的是从格林尼治标准时间 1970 年 1 月 1 日 0 时起到现在的秒数。这个值被转换成 unsigned 类型的整数，并用做随机数生成器中的种子。time 的函数原型在<ctime>头文件中。



常见的编程错误 6.7

在程序中多次调用 srand 函数会重启伪随机数字序列，并会影响 rand 生成的数字的随机性。

比例缩放和偏移生成随机数的概括

前面，我们演示了如何通过一条简单语句来模拟掷六面骰子，该语句如下所示：

```
face = 1 + rand() % 6 ;
```

该语句总是将一个整数值（随机）赋给取值范围在 1 ≤ face ≤ 6 之间的变量 face。请注意这一范围的宽度（即范围中连续整数的个数）是 6，起始数字是 1。参照前面的语句，我们可以看到范围的宽度由用取模运算符对 ramd 进行比例缩放所用的数（即 6）决定，并且范围的起始数字等于添加到表达式 rand % 6 中的数字（也就是 1）。我们可以对这个结果进行如下的概括：

```
number = shiftingValue + rand() % scalingFactor;
```

这里shiftingValue相当于所需的连续整数范围内的第一个数字，scalingFactor相当于所需的连续整数范围的宽度。习题表明是可以从一组非连续整数中随机选择整数的。



常见的编程错误 6.8

试图使用 srand 代替 rand 函数来生成随机数会导致编译错误——srand 函数并没有返回值。

```
// Fig. 6.10: fig06_10.cpp
// Randomizing die-rolling program.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib> // contains prototypes for functions srand and rand
using std::rand;
using std::srand;

int main()
{
   unsigned seed; // stores the seed entered by the user

   cout << "Enter seed: ";
   cin >> seed;
   srand( seed ); // seed random number generator

   // loop 10 times
   for ( int counter = 1; counter <= 10; counter++ )
   {
      // pick random number from 1 to 6 and output it
      cout << setw( 10 ) << ( 1 + rand() % 6 );

      // if counter is divisible by 5, start a new line of output
      if ( counter % 5 == 0 )
         cout << endl;
   } // end for

   return 0; // indicates successful termination
} // end main
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

```
Enter seed: 432
         4         6         3         1         6
         3         1         5         4         2
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

图 6.10　随机化掷骰子程序

6.8　实例研究：博彩游戏和 enum 简介

在最流行的博彩游戏中有一种名为“掷双骰”（craps）的骰子游戏，这种游戏是在娱乐场玩的，并且在全世界风靡流行。游戏的规则很简单：

> 玩家掷两个骰子。每个骰子有六面，分别含有1、2、3、4、5和6个点。掷完骰子后，就计算两个朝上的面的点数之和。如果首次投掷的点数总和是7或者11的话，那么玩家赢；如果首次投掷的点数之和是2、3或者12的话（称为“craps”），那么玩家输（即庄家赢）；如果首次投掷的点数之和是4、5、6、8、9或者10，那么这个和就成为玩家的“点数”。要想赢的话，玩家必须连续地掷骰子直到点数同这个目标点数相同为止。即“得到了点数”。但在得到点数前，如果掷到的是7，就会输掉。

图 6.11 中的程序模拟了掷双骰游戏。

```
1  // Fig. 6.11: fig06_11.cpp
2  // Craps simulation.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstdlib> // contains prototypes for functions srand and rand
8  using std::rand;
9  using std::srand;
10
11 #include <ctime> // contains prototype for function time
12 using std::time;
13
14 int rollDice(); // rolls dice, calculates amd displays sum
15
16 int main()
17 {
18    // enumeration with constants that represent the game status
19    enum Status { CONTINUE, WON, LOST }; // all caps in constants
20
21    int myPoint; // point if no win or loss on first roll
22    Status gameStatus; // can contain CONTINUE, WON or LOST
23
24    // randomize random number generator using current time
25    srand( time( 0 ) );
26
27    int sumOfDice = rollDice(); // first roll of the dice
28
29    // determine game status and point (if needed) based on first roll
30    switch ( sumOfDice )
31    {
32       case 7: // win with 7 on first roll
33       case 11: // win with 11 on first roll
34          gameStatus = WON;
35          break;
36       case 2: // lose with 2 on first roll
37       case 3: // lose with 3 on first roll
38       case 12: // lose with 12 on first roll
39          gameStatus = LOST;
40          break;
41       default: // did not win or lose, so remember point
42          gameStatus = CONTINUE; // game is not over
43          myPoint = sumOfDice; // remember the point
44          cout << "Point is " << myPoint << endl;
45          break; // optional at end of switch
46    } // end switch
47
```

图 6.11　掷双骰游戏的模拟

```
   // while game is not complete
   while ( gameStatus == CONTINUE ) // not WON or LOST
   {
      sumOfDice = rollDice(); // roll dice again

      // determine game status
      if ( sumOfDice == myPoint ) // win by making point
         gameStatus = WON;
      else
         if ( sumOfDice == 7 ) // lose by rolling 7 before point
            gameStatus = LOST;
   } // end while

   // display won or lost message
   if ( gameStatus == WON )
      cout << "Player wins" << endl;
   else
      cout << "Player loses" << endl;

   return 0; // indicates successful termination
} // end main

// roll dice, calculate sum and display results
int rollDice()
{
   // pick random die values
   int die1 = 1 + rand() % 6; // first die roll
   int die2 = 1 + rand() % 6; // second die roll

   int sum = die1 + die2; // compute sum of die values

   // display results of this roll
   cout << "Player rolled " << die1 << " + " << die2
      << " = " << sum << endl;
   return sum; // end function rollDice
} // end function rollDice
```

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```

图 6.11 掷双骰游戏的模拟（续）

```
1  // Fig. 6.12: fig06_12.cpp
2  // A scoping example.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void useLocal( void ); // function prototype
8  void useStaticLocal( void ); // function prototype
9  void useGlobal( void ); // function prototype
10
11 int x = 1; // global variable
12
13 int main()
14 {
15    int x = 5; // local variable to main
16
17    cout << "local x in main's outer scope is " << x << endl;
18
19    { // start new scope
20       int x = 7; // hides x in outer scope
21
22       cout << "local x in main's inner scope is " << x << endl;
23    } // end new scope
24
25    cout << "local x in main's outer scope is " << x << endl;
26
27    useLocal(); // useLocal has local x
28    useStaticLocal(); // useStaticLocal has static local x
29    useGlobal(); // useGlobal uses global x
30    useLocal(); // useLocal reinitializes its local x
31    useStaticLocal(); // static local x retains its prior value
32    useGlobal(); // global x also retains its value
33
34    cout << "\nlocal x in main is " << x << endl;
35    return 0; // indicates successful termination
36 } // end main
37
38 // useLocal reinitializes local variable x during each call
39 void useLocal( void )
40 {
41    int x = 25; // initialized each time useLocal is called
42
43    cout << "\nlocal x is " << x << " on entering useLocal" << endl;
44    x++;
45    cout << "local x is " << x << " on exiting useLocal" << endl;
46 } // end function useLocal
47
48 // useStaticLocal initializes static local variable x only the
49 // first time the function is called; value of x is saved
50 // between calls to this function
51 void useStaticLocal( void )
52 {
53    static int x = 50; // initialized first time useStaticLocal is called
54
55    cout << "\nlocal static x is " << x << " on entering useStaticLocal"
56       << endl;
57    x++;
58    cout << "local static x is " << x << " on exiting useStaticLocal"
59       << endl;
60 } // end function useStaticLocal
61
62 // useGlobal modifies global variable x during each call
63 void useGlobal( void )
64 {
65    cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
66    x *= 10;
67    cout << "global x is " << x << " on exiting useGlobal" << endl;
```

图 6.12　作用域示例

```
68  } // end function useGlobal

local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

图 6.12 作用域示例（续）

为了演示其他的作用域，程序定义了三个函数，每个函数都是没有参数和返回值的。函数 useLocal（第 39 ~ 46 行）声明了自动变量 x（第 41 行）并初始化为 25。当程序调用 useLocal 函数时，该函数打印这个变量，递增它的值并在函数将程序控制数返回给其调用者之前再次打印它。程序每次调用这个函数时，函数都重新创建自动变量 x 并重新将之初始化为 25。

useStaticLocal 函数（第 51 ~ 60 行）声明了 static 变量 x 并将它初始化为 50。声明为 static 的局部变量甚至在它们的作用域之外（即声明它们的函数不再执行时）仍保留其值。当程序调用 useStaticLocal 时，函数打印出 x，增 x 的值并在函数将程序控制权返回给调用者前再次打印 x 的值。下次再调用这个函数时，static 局部变量 x 的值为 51。第 53 行的初始化只发生一次——在第一次调用 useStaticLocal 时。

useGlobal 函数（第 63 ~ 68 行）没有声明任何变量。因此，当引用变量 x 时用的是全局变量 x（在 main 之前定义的）。当程序调用 useGlobal 时，函数打印出全局变量 x 的值，将其乘 10 并在函数把程序控制权返回给调用者之前再次打印其值。在程序下次调用 useGlobal 时，这个全局变量具有已修改后的值 10。执行函数 useLocal、useStaticLocal 和 useGlobal 各两次之后，程序再次打印在 main 函数中的局部变量 x 的值，以表明没有哪个函数调用修改了在 main 中的 x 的值，因为所有这些函数都是引用其他作用域的变量。

6.11 函数调用堆栈和活动记录

为了理解 C++ 如何执行函数调用，我们先来看看称为堆栈（stack）的数据结构（即相关数据项的集合）。可以把堆栈比成一叠盘子。当一个盘子放到一叠盘子上时，通常是放在最上面［将此过程称做把盘子压到（pushing）堆栈中）］。类似地，当从一叠盘子中取一个盘子时，通常是从最顶部取［将此过程称做把盘子从堆栈中弹出（popping）］。堆栈被称为是后进先出（LIFO，last-in, first-out）的数据结构——最后压到（插入到）堆栈中的项最先从堆栈中弹出（移除）。

主修计算机科学的学生需要理解的最重要的机制之一是函数调用堆栈（function call stack），有时也称为程序执行堆栈（program execution stack）。这种数据结构（在后台工作）用于支持函数调用/返回机制。堆栈也支持每个被调用函数的自动变量的创建、维护和销毁。我们通过一叠盘子的例子解释了堆栈的后进先出（LIFO）特性。如图 6.14 至图 6.16 所示，函数在返回到调用它的函数时就是符合这种后进先出的特性的。

调用每个函数时，可能会依次调用其他函数，所调用的函数也可能再调用其他函数——这一切都发生在任何函数返回之前。每个函数最终都必须把控制权返回给调用它的函数。因此，我们必须用某种方法记录每个函数需要返回给其调用者控制权的返回地址。函数调用堆栈是处理这些信息的理想数据结构。每次函数调用另一个函数时，一个数据项会压入到堆栈中。这一数据项，称为一个堆栈结构（stack frame）或者一条活动记录（activation record），包含了被调函数返回到其调用者所需的返回地址。还包含我们即将讨论的一些附加信息。如果被调用的函数返回，而不是在返回前调用其他函数，那么这个函数调用的堆栈结构就会弹出，并且控制权会传给该结构中的返回地址。

调用堆栈的美妙之处在于每个被调用的函数总是能在调用堆栈的最顶部找到它需要返回给调用者的信息。而且，如果一个函数调用另一个函数，新的函数调用的堆栈结构只要简单地压到调用堆栈上即可。这样，新的被调用函数所要的返回到调用者的返回地址就在堆栈的最顶部了。

堆栈结构还有另一个重要的责任。大多数函数都有自动变量——参数和函数中声明的任何局部变量。自动变量在函数执行时应当存在。如果函数调用其他函数，它们需要处于活动状态。但是当被调用函数的返回到调用者时，被调用函数的自动变量需要“销毁”。被调用函数的堆栈结构是保留被调用函数自动变量所用内存的理想地方。只要被调用函数还在执行，对应的堆栈结构就一直存在。当被调用函数返回，并且不再需要它的局部自动变量时，它的堆栈结构从堆栈中弹出，然后程序就不再知道这些局部自动变量了。

当然，计算机的内存大小是有限的，因此只能有一定量的内存可用于保存函数调用堆栈的活动记录。如果太多的函数调用发生，以至于不能把相应的活动记录保存到函数调用堆栈中，就会发生堆栈溢出（stack overflow）错误。

活动中的函数调用堆栈

我们知道，调用堆栈和活动记录支持函数调用/返回机制和自动变量的创建与销毁。现在让我们来看看调用堆栈是如何支持由 main 函数（见图 6.13 中的第 11 ~ 17 行）调用的 square 函数的操作的。首先，操作系统调用 main——这会把活动记录压到堆栈上（如图 6.14 所示）。这条活动记录告诉 main 如何返回到操作系统（即转到返回地址 R1）并且包含 main 的自动变量（即初始化为 10 的 a）的空间。

```
1  // Fig. 6.13: fig06_13.cpp
2  // square function used to demonstrate the function
3  // call stack and activation records.
4  #include <iostream>
5  using std::cin;
6  using std::cout;
7  using std::endl;
8
9  int square( int ); // prototype for function square
10
11 int main()
12 {
13    int a = 10; // value to square (local automatic variable in main)
14
15    cout << a << " squared: " << square( a ) << endl; // display a squared
16    return 0; // indicate successful termination
17 } // end main
18
19 // returns the square of an integer
20 int square( int x ) // x is a local variable
21 {
22    return x * x; // calculate square and return result
23 } // end function square
```

```
10 squared: 100
```

图 6.13　使用 square 函数演示函数调用堆栈和活动记录

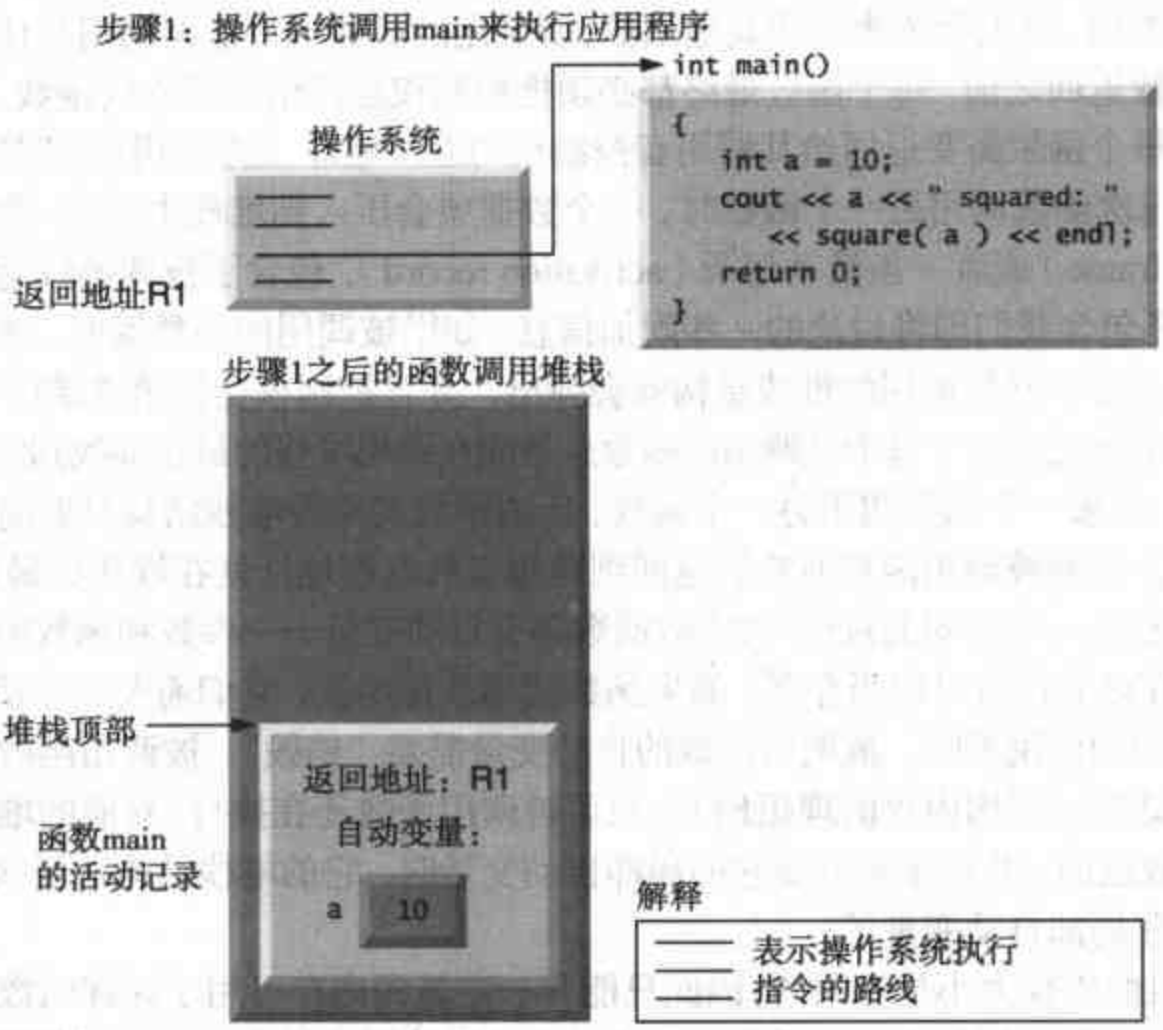


图 6.14　在系统调用 main 函数执行应用程序之后的函数调用堆栈

在返回到操作系统之前，函数 main 现在调用图 6.13 中第 15 行的函数 square。这样 square（第20～23行）函数的堆栈结构就被压入到函数调用堆栈中（见图6.15）。这个堆栈结构包含了square需要返回到 main 的返回地址（即 R2）和 square 的自动变量（即 x）的内存。

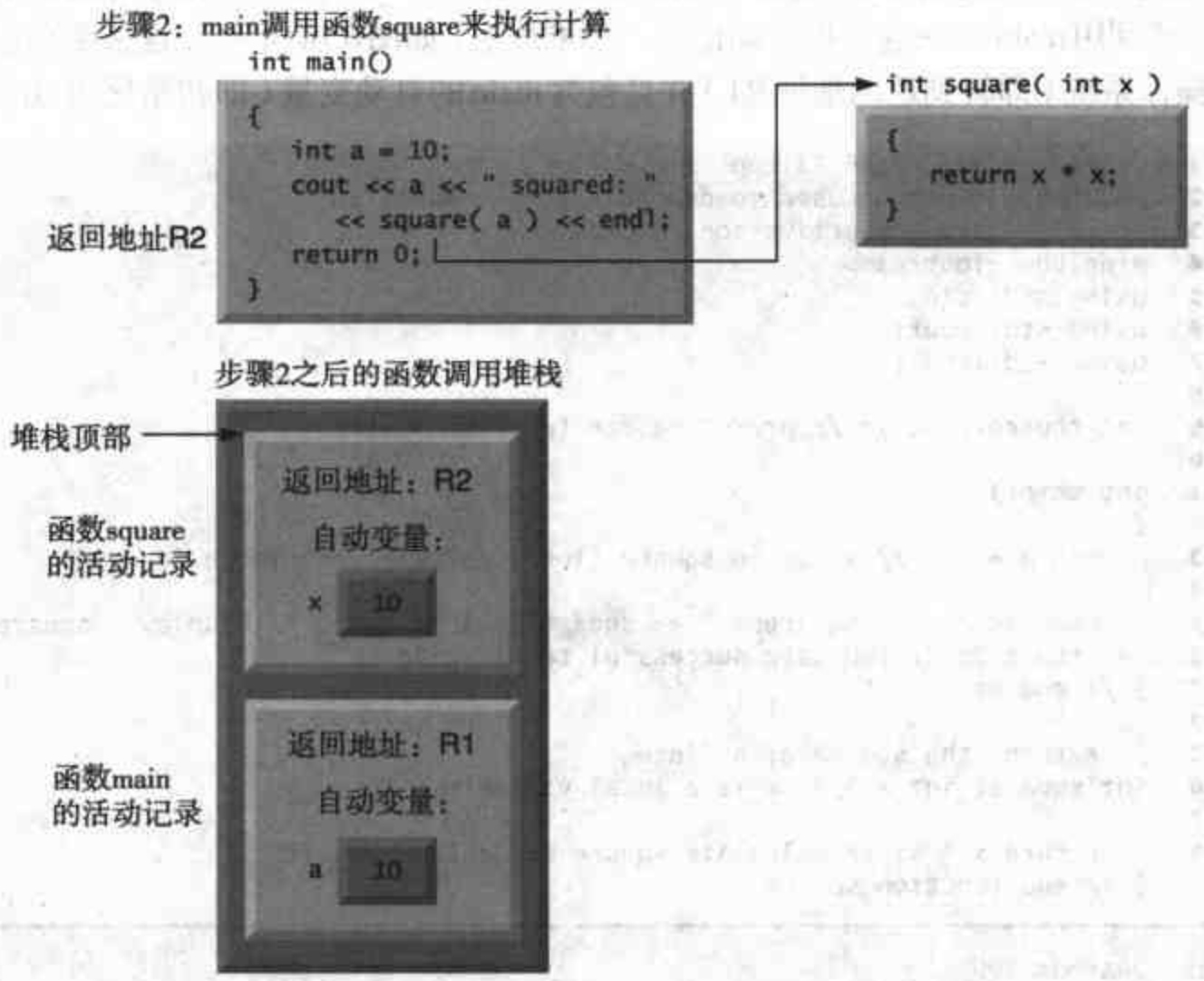


图 6.15　在 main 调用函数 square 执行计算之后的函数调用堆栈

在 square 函数计算完其参数的平方后，它需要返回到 main 函数——并且不再需要自动变量 x 的内存。因此堆栈弹出——给出 square 返回到 main 函数的地址（即 R2）并丢弃 square 的自动变量。图 6.16 显示了 square 的活动记录弹出后的函数调用堆栈。

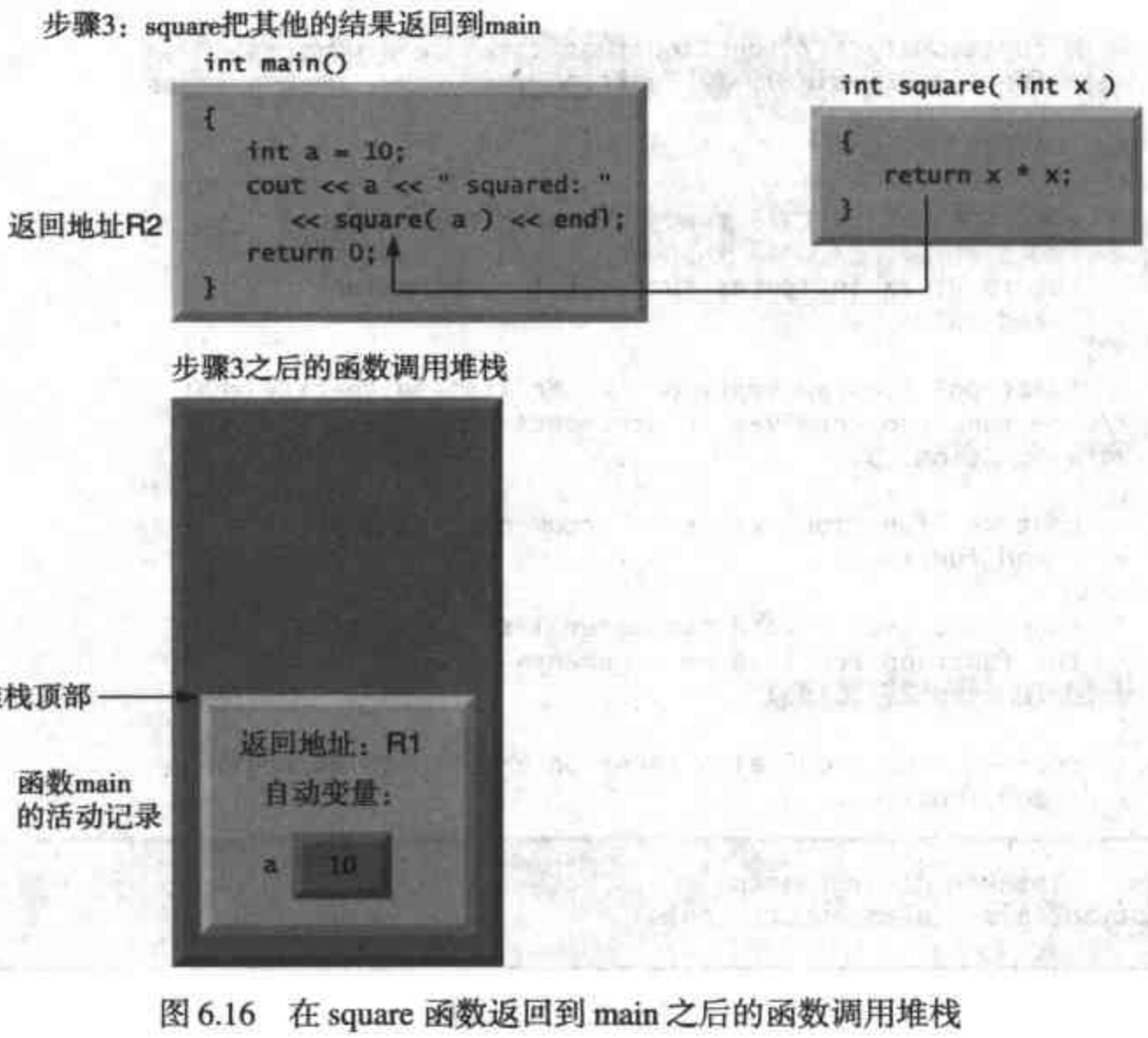


图 6.16　在 square 函数返回到 main 之后的函数调用堆栈

现在 main 函数显示调用 square 的结果（第 15 行），然后执行 return 语句（第 16 行）。这会使得 main 函数的活动记录从堆栈中弹出。弹出的记录给了 main 函数返回到操作系统的地址（即图 6.14 中的 R1）并使得 main 函数中的自动变量（即 a）的内存不再有效。

读者现在看到了在实现支持程序执行的关键机制中堆栈数据结构概念的价值。在计算机科学中数据结构具有许多重要的应用。我们将在本书的姐妹篇《C++ 大学教程》（第五版）的第 21 章和第 23 章中讨论堆栈、队列、链表、树以及其他数据结构。

6.12　无参函数

在 C++ 中，空的参数列表可以通过在圆括号里面写 void 或者什么都不写来表示。如下的原型：

```
void print();
```

明确说明 print 函数不接收任何参数并且不返回任何值。图 6.17 展示了声明和使用参数表为空的函数的两种方法。



可移植性提示 6.2

C++ 中空的函数参数列表的含义与 C 语言中的有很大不同。在 C 中，空的参数列表意味着不进行任何参数检查（即函数调用可以传递它想传递的任何参数）；在 C++ 中，空的参数列表则意味着函数明确地不需要任何参数。因此，使用该特性的 C 程序在 C++ 中编译可能会引起编译错误。

```
1  // Fig. 6.17: fig06_17.cpp
2  // Functions that take no arguments.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void function1(); // function that takes no arguments
8  void function2( void ); // function that takes no arguments
9
10 int main()
11 {
12    function1(); // call function1 with no arguments
13    function2(); // call function2 with no arguments
14    return 0; // indicates successful termination
15 } // end main
16
17 // function1 uses an empty parameter list to specify that
18 // the function receives no arguments
19 void function1()
20 {
21    cout << "function1 takes no arguments" << endl;
22 } // end function1
23
24 // function2 uses a void parameter list to specify that
25 // the function receives no arguments
26 void function2( void )
27 {
28    cout << "function2 also takes no arguments" << endl;
29 } // end function2
```

```
function1 takes no arguments
function2 also takes no arguments
```

图 6.17 无参函数



常见的编程错误 6.13

C++程序不会编译成功，除非每个函数都提供了函数原型或者在函数被调用之前已经定义了。

6.13 内联函数

从软件工程的立场来看，把程序通过一组函数来实现是不错的方法，但是函数调用涉及执行时间的开销。C++提供了内联函数（inline function）来减少函数调用的开销——特别是对于小函数。在函数定义中把限定词inline放在函数的返回类型的前面会“建议”编译器在适当的地方生成函数代码的副本以避免函数调用。这种折中办法是把函数代码的多份副本插入到程序中（通常会使程序变大），而不是仅有一份函数代码，每次调用函数时都把控制权传递给它。编译器可以忽略inline限定词，并且对于除了小函数外的函数，通常都会这样做。



软件工程知识 6.11

对内联函数所做的任何修改要求该函数的所有客户重新进行编译。这在程序开发和维护中是非常重要的。



良好的编程习惯 6.5

inline限定词应该只适用于小的、经常使用的函数。



性能提示 6.4

使用内联函数可以减少执行时间，但是会增加程序尺寸。

图6.18使用了内联函数cube（第11～14行）来计算一个边长为side的立方体的体积。函数cube的参数列表中的关键字const（第11行）告诉编译器函数不修改变量side的值。这就保证了在计算执行时函数不会修改side的值（关键字const在第7章、第8章和第10章中详细讨论。）请注意程序中cube函数的完整定义出现在该函数的使用之前。这样做是必需的，它使得编译器知道如何把cube函数调用扩展为它的内联代码。基于这样的原因，可重用的内联函数一般都放在头文件中，以便使用它们的每个源文件都能包含它们的定义。

```
1  // Fig. 6.18: fig06_18.cpp
2  // Using an inline function to calculate the volume of a cube.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  // Definition of inline function cube. Definition of function appears
9  // before function is called, so a function prototype is not required.
10 // First line of function definition acts as the prototype.
11 inline double cube( const double side )
12 {
13    return side * side * side; // calculate cube
14 } // end function cube
15
16 int main()
17 {
18    double sideValue; // stores value entered by user
19    cout << "Enter the side length of your cube: ";
20    cin >> sideValue; // read value from user
21
22    // calculate cube of sideValue and display result
23    cout << "Volume of cube with side "
24       << sideValue << " is " << cube( sideValue ) << endl;
25    return 0; // indicates successful termination
26 } // end main
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

图6.18　计算立方体体积的inline函数



软件工程知识 6.12

应该利用const限定词增强最小特权原则。使用最小特权原则进行正确的软件设计可以大大减少调试时间和错误的副作用，也使得程序更容易修改和维护。

6.14　引用和引用参数

在许多编程语言中都有按值传递（pass-by-value）和按引用传递（pass-by-reference）这两种函数参数传递方式。当用按值传递的方式传递参数时，会（在函数调用堆栈上）产生一份实参值的副本，然后将副本传递给被调用的函数。对副本的修改不影响调用者中原始变量的值。这样就防止了对正确可靠的软件系统的开发产生很大阻碍的副作用。到目前为止，本章在程序中传递的每个实参都是按值传递的。



性能提示 6.5

按值传递的一个缺点是，如果有一个大的数据项需要传递，那么复制这些数据就需要花费大量的时间和内存空间。

引用参数

本节介绍引用参数（reference parameter）——C++提供的执行按引用传递的两种方法中的第一种。利用按引用传递，调用者使得被调用函数可以直接访问调用者的数据，并且如果被调用函数需要的话，还可以修改这些数据。



性能提示 6.6

按引用传递对性能很有帮助，因为它可以消除按值传递复制大量数据的开销。



软件工程知识 6.13

按引用传递可能削弱安全性，因为被调用函数可能破坏调用者的数据。

后面我们将说明如何在获得保护调用者的数据不被破坏的软件工程优势的同时，实现按引用传递的性能优势。

引用参数是函数调用中相应实参的别名。为了指明一个函数参数是按引用传递的，只要简单地在函数原型中参数类型后加一个 & 标识即可；在函数头部列出参数类型时也采用相同的做法。例如，下面的在函数头部中的声明：

```
int &count
```

从右到左读做“count 是对一个 int 类型对象的引用”。在函数调用中，简单地提及按引用传递的变量名，然后，在被调用函数的体内函数参数名代表的变量实际上引用的是调用函数中的原始变量，并且被调用函数可以直接修改原始变量。像以往一样，函数原型和函数头部必须保持一致。

按值传递和按引用传递实参

图 6.19 比较了按值传递和带引用参数的按引用传递。在函数 squareByValue 和 squareByReference 的调用中，实参“形式”是相同的——只需在函数调用中简单地加入变量名即可。如果不检查函数原型或者函数定义，就不可能由函数调用判断出哪个函数可以修改它的实参。然而，因为函数原型是强制性的，编译器可以轻而易举地解决这种歧义性问题。

```
1  // Fig. 6.19: fig06_19.cpp
2  // Comparing pass-by-value and pass-by-reference with references.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int squareByValue( int ); // function prototype (value pass)
8  void squareByReference( int & ); // function prototype (reference pass)
9
10 int main()
11 {
12    int x = 2; // value to square using squareByValue
13    int z = 4; // value to square using squareByReference
14
15    // demonstrate squareByValue
16    cout << "x = " << x << " before squareByValue\n";
17    cout << "Value returned by squareByValue: "
18       << squareByValue( x ) << endl;
19    cout << "x = " << x << " after squareByValue\n" << endl;
20
21    // demonstrate squareByReference
22    cout << "z = " << z << " before squareByReference" << endl;
23    squareByReference( z );
24    cout << "z = " << z << " after squareByReference" << endl;
25    return 0; // indicates successful termination
```

图 6.19　按值传递和按引用传递参数

```
26  } // end main
27
28  // squareByValue multiplies number by itself, stores the
29  // result in number and returns the new value of number
30  int squareByValue( int number )
31  {
32     return number *= number; // caller's argument not modified
33  } // end function squareByValue
34
35  // squareByReference multiplies numberRef by itself and stores the result
36  // in the variable to which numberRef refers in function main
37  void squareByReference( int &numberRef )
38  {
39     numberRef *= numberRef; // caller's argument modified
40  } // end function squareByReference
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

图6.19　按值传递和按引用传递参数（续）



常见的编程错误 6.14

因为在被调用函数体中引用参数只需要用名字来指定，程序员可能在不经意间把引用参数当成按值传递的参数。这样的话，如果函数改变了原始的变量副本，就会产生不可预期的副作用。

第8章将讨论指针；指针会提供另一种形式的按引用传递，在这种形式中调用形式非常清晰地指明了按引用传递（并且指明了可能会修改调用者的参数）。



性能提示 6.7

为了传递大型对象，使用一个常量引用参数来模拟按值传递的外观和安全性，并且避免传递大型对象的副本的开销。



软件工程知识 6.14

许多程序员不愿意把按值传递的参数声明为const，即使被调用函数不修改传递的实参。在这种情况下，关键字const将只保护原始实参的副本，而不是原始实参本身，因此在按值传递时原始实参对于被调用函数所做的修改来说是安全的。

为了指定常量的引用，在参数声明的类型说明符前加上const限定词就可以了。

请注意图6.19的第37行中squareByReference函数的参数列表里放置的&。有些程序员更喜欢使用int& numberRef。



软件工程知识 6.15

为了清晰和良好性能起见，许多C++程序员宁愿使用指针（我们将在第8章中学习）来向函数传递可修改的实参，而小的不可修改的参数采用值传递，大的不可修改的参数通过使用常量的引用来传递给函数。

在函数内引用作为别名

引用还可以在函数中用做其他变量的别名（虽然它们通常都像图6.19中所示的那样与函数一起使用）。例如，代码：

```
int count = 1 ; // declare integer variable count
int &cRef = count; // create cRef as an alias for count
cRef++; // increment count (using its alias cRef)
```

通过使用变量 count 的别名 cRef 来自增变量 count。引用变量必须在它们的声明中完成初始化（见图 6.20 和图 6.21）并且不能再指定为其他变量的别名。一旦一个引用被声明为另一个变量的别名，在别名（即引用）上执行的所有操作实际上作用在原始变量上。别名只是简单地作为原始变量的另一个名字。获取引用的地址和比较引用不会产生语法错误；相反，每个操作实际上都发生在引用作为别名的变量上。除非是对常量的引用，否则引用参数必须是左值（例如，变量名），而不是常量或者返回右值的表达式（例如，计算结果）。关于左值和右值的定义见 5.9 节。

```
1  // Fig. 6.20: fig06_20.cpp
2  // References must be initialized.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 3;
10    int &y = x; // y refers to (is an alias for) x
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7; // actually modifies x
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indicates successful termination
16 } // end main
```

```
x = 3
y = 3
x = 7
y = 7
```

图 6.20　初始化和使用引用

从函数返回引用

函数可以返回引用，但是这种方法可能存在危险。当返回一个在被调用函数中声明的变量的引用时，这个变量应该在函数中声明为 static 的。否则，该引用指的是在函数执行结束时被销毁的自动变量，这样的变量会被告知“未定义”并且程序的行为会不可预知。对一个未定义的变量的引用称为虚悬引用（dangling reference）。



常见的编程错误 6.15

在声明引用变量时不初始化它会产生编译错误，除非此声明是函数参数列表的一部分。当引用参数声明所在的函数被调用时它们被初始化。



常见的编程错误 6.16

试用把以前已声明的引用再作为另一个变量的别名是一种逻辑错误。另一个变量的值只赋给引用已是其别名的变量。



常见的编程错误 6.17

返回被调用函数中的自动变量的引用会产生逻辑错误。有些编译器会在这种逻辑错误发生时产生警告。

未初始化引用的错误消息

请注意 C++ 标准没有指定编译器用于显示特定错误的相应的错误信息。基于这个原因，图 6.21 所示的是在引用未初始化时 Borland C++ 5.5 命令行编译器、Microsoft Visual C++.NET 编译器和 GNU C++ 编译器生成的错误信息。

```
1  // Fig. 6.21: fig06_21.cpp
2  // References must be initialized.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 3;
10    int &y; // Error: y must be initialized
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7;
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indicates successful termination
16 } // end main
```

Borland C++ command-line compiler error message:

```
Error E2304 C:\scpphtp5_examples\ch06\Fig06_21\fig06_21.cpp 10:
   Reference variable 'y' must be initialized in function main()
```

Microsoft Visual C++ compiler error message:

```
C:\scpphtp5_examples\ch06\Fig06_21\fig06_21.cpp(10) : error C2530: 'y' :
   references must be initialized
```

GNU C++ compiler error message:

```
fig06_21.cpp:10: error: 'y' declared as a reference but not initialized
```

图 6.21　未初始化的引用引起的语法错误

6.15　默认实参

程序中这样的现象并不少见，即重复调用函数时对某个特定的形参一直用相同的实参。在这种情况下，程序员可以对这样的形参指定默认实参（default argument），即传递给该形参的一个默认的值。当一个程序在函数调用中对于只有默认实参的形参省略了其对应的实参时，编译器重写这个函数调用并且插入那个实参的默认值作为函数调用所传递的实参。

默认实参必须是函数参数列表中最靠右边（尾部）的参数。当调用具有两个或者更多个默认实参的函数时，如果省略的实参不是参数列表中最靠右边的参数，那么该实参右边的所有参数也必须被省略。默认实参应该在函数名第一次出现时指定——通常是在函数原型中。如果因为函数定义也作为函数原型而省略了函数原型，那么默认实参应该在函数头部中指定。默认值可以是任何表达式，包括常量、全局变量或者函数调用。默认实参也能用于内联函数。

图6.22演示了使用默认实参计算盒子的体积。boxVolume的函数原型（第8行）指定三个形参的默认值都为1。请注意我们在函数原型中提供变量名是为了增加可读性。通常，变量名在函数原型中并不需要。



常见的编程错误 6.18
在函数原型和函数头部中同时指定默认实参会产生编译错误。

第一次调用boxVolume 函数（第13行）没有指定任何参数，因此使用的是三个默认值为1的参数。第二次调用（第17行）向length传递了一个实参，这样就使用了width和height的默认值1。

第三次调用（第21行）传递了给 length 和 width 的实参，这样使用的是 height 为 1 的默认值。最后一次调用（第25行）向 length、width 和 height 传递了实参，这样就没使用默认值。请注意显式地传递给函数的任何实参都按从左到右的顺序赋给函数的形参。因此，当 boxVolume 接受一个实参时，函数把那个实参的值赋给它的 length 形参（即参数列表中最左边的形参）。当 boxVolume 接受两个参数时，函数把这两个实参的值按照顺序分别赋给它的 length 和 width 形参。最后，当 boxVolume 接受全部三个实参时，函数把这些实参的值分别赋给 length、width 和 height 形参。

```
1  // Fig. 6.22: fig06_22.cpp
2  // Using default arguments.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // function prototype that specifies default arguments
8  int boxVolume( int length = 1, int width = 1, int height = 1 );
9
10 int main()
11 {
12    // no arguments--use default values for all dimensions
13    cout << "The default box volume is: " << boxVolume();
14
15    // specify length; default width and height
16    cout << "\n\nThe volume of a box with length 10,\n"
17       << "width 1 and height 1 is: " << boxVolume( 10 );
18
19    // specify length and width; default height
20    cout << "\n\nThe volume of a box with length 10,\n"
21       << "width 5 and height 1 is: " << boxVolume( 10, 5 );
22
23    // specify all arguments
24    cout << "\n\nThe volume of a box with length 10,\n"
25       << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
26       << endl;
27    return 0; // indicates successful termination
28 } // end main
29
30 // function boxVolume calculates the volume of a box
31 int boxVolume( int length, int width, int height )
32 {
33    return length * width * height;
34 } // end function boxVolume
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

图 6.22　函数的默认实参



良好的编程习惯 6.6

使用默认实参可以简化函数调用的书写。然而，有些程序员觉得显式地指定所有的实参更加清晰。



软件工程知识 6.16

如果函数的默认值改变了，所有使用该函数的客户代码必须重新编译。



常见的编程错误 6.19
对非最右端的参数指定和尝试使用默认实参（当没有同时默认所有最右端的实参时）会产生语法错误。

6.16　一元作用域分辨运算符

有可能局部变量和全局变量声明为相同的名字。C++提供了一元作用域分辨运算符（::），以便当作用域中的局部变量与全局变量具有相同的名字时访问全局变量。不能使用一元作用域分辨运算符访问外层语句块中具有相同名字的局部变量。如果全局变量和作用域中的局部变量的名字不同，那么不用一元作用域分辨运算符就可以直接访问全局变量。

图 6.23 所示的是局部变量和全局变量名字相同（第 7 和 11 行）时使用一元作用域分辨运算符的情况。为了强调变量 number 的局部变量版和全局变量版是不同的，程序声明一个变量为 int 类型而另一个变量为 double 类型。

```
1  // Fig. 6.23: fig06_23.cpp
2  // Using the unary scope resolution operator.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int number = 7; // global variable named number
8
9  int main()
10 {
11    double number = 10.5; // local variable named number
12
13    // display values of local and global variables
14    cout << "Local double value of number = " << number
15       << "\nGlobal int value of number = " << ::number << endl;
16    return 0; // indicates successful termination
17 } // end main
```

```
Local double value of number = 10.5
Global int value of number = 7
```

图 6.23　一元作用域分辨运算符

当具有某个名字的惟一变量是一个全局变量时，对这个给定的变量名使用一元作用域分辨运算符（::）是可任意选择的。



常见的编程错误 6.20
试图使用一元作用域分辨运算符（::）访问外层语句块中非全局变量是错误的。如果给定名字的全局变量不存在，就会发生编译错误。如果给定名字的全局变量存在，那么就是逻辑错误，因为在你打算访问外层语句块中的非全局变量时程序引用的是全局变量。



良好的编程习惯 6.7
总是使用一元作用域分辨运算符（::）来引用全局变量会使得程序更易于阅读和理解，因为可以更加清楚地知道用户需要访问的是一个全局变量而不是非全局变量。



软件工程知识 6.17
总是使用一元作用域分辨运算符（::）引用全局变量降低了与非全局变量名字冲突的风险，使得程序更易于修改。



错误预防技巧 6.4
总是使用一元作用域分辨运算符（::）引用全局变量消除了如果全局变量隐藏全局变量可能发生的逻辑错误。



错误预防技巧 6.5
在程序中避免给用于不同目的的变量使用相同的名字。虽然在不同情况下这是允许的，但是它会导致错误。

6.17 函数重载

C++ 允许定义多个具有相同名字的函数，只要这些函数的参数不同（至少参数类型或者参数数目或者参数类型的顺序不同）。这种特性称为函数重载。当调用一个重载函数时，C++ 编译器通过检查函数调用中的实参的数目、类型和顺序来选择恰当的函数。函数重载通常用于创建执行相似任务、但是作用于不同的数据类型的具有相同名字的多个函数。例如，数学库中的许多函数对于不同的数字数据类型是重载的[①]。



良好的编程习惯 6.8
重载执行紧密相关任务的函数可以使程序更易阅读和理解。

重载的 square 函数

图 6.24 中使用了重载的 square 函数来分别计算一个 int 类型值（第 8 ~ 12 行）和一个 double 类型值（第 15 ~ 19 行）的平方。第 23 行通过传递一个字面值 7 来调用 int 版本的 square 函数，默认时 C++ 把整个数字字面值当做 int 类型的。类似地，第 25 行通过传递字面值 7.5 来调用 double 版本的 square 函数，默认时 C++ 把字面值当做 double 值。在两种情形中编译器会根据实参类型选择恰当的函数来调用。输出窗口的最后两行证实了在每种情形下调用了适当的函数。

```
1  // Fig. 6.24: fig06_24.cpp
2  // Overloaded functions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // function square for int values
8  int square( int x )
9  {
10    cout << "square of integer " << x << " is ";
11    return x * x;
12 } // end function square with int argument
13
14 // function square for double values
15 double square( double y )
16 {
17    cout << "square of double " << y << " is ";
18    return y * y;
19 } // end function square with double argument
20
21 int main()
22 {
23    cout << square( 7 ); // calls int version
24    cout << endl;
25    cout << square( 7.5 ); // calls double version
26    cout << endl;
27    return 0; // indicates successful termination
28 } // end main
```

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

图 6.24 重载的 square 函数

① 对于 6.3 节所述的数学库函数，C++ 标准需要 float、double 和 long double 的重载版本。

编译器如何区分重载的函数

重载的函数通过它们的签名来区分。签名由函数的名字和它的参数类型（按顺序）组成。编译器对每个函数的标识符利用它的参数数目和类型进行编码，以便能够实现类型安全的连接。类型安全的连接保证调用正确的重载函数并且实参类型与形参类型相符合。

图 6.25 显示的是用 Borland C++ 5.6.4 命令行编译器编译的程序。这里没有显示程序的执行结果（通常我们会这样做），而是显示了由 Borland C++ 用汇编语言生成的改编后的函数名。每个改编后的名称以@开始，后跟函数名；然后函数名通过$q与改编后的参数列表相分离。在函数nothing2的参数列表中（第 25 行；见输出的第 4 行），c 表示 char，i 表示 int，rf 表示 float &（即，float 的引用），rd 表示 double&（即 double 的引用）。在函数 nothing1 的参数列表中，i 表示 int，f 表示 float，c 表示 char，ri 表示 int&。两个 square 函数通过它们的参数列表来区分；一个将 double 指定为 d，而另一个将 int 指定为 i。函数的返回类型在改编后的名称中没有指定。重载的函数可以有不同的返回类型，如果返回类型相同，那么它们必须有不同的参数列表。是的，你不能把两个函数写成具有相同的签名，不同的只是返回类型。注意函数名称改编是与编译器相关的。还要注意的是，main 函数不能改编，因为它不能被重载。

```
1  // Fig. 6.25: fig06_25.cpp
2  // Name mangling.
3
4  // function square for int values
5  int square( int x )
6  {
7     return x * x;
8  } // end function square
9
10 // function square for double values
11 double square( double y )
12 {
13    return y * y;
14 } // end function square
15
16 // function that receives arguments of types
17 // int, float, char and int &
18 void nothing1( int a, float b, char c, int &d )
19 {
20    // empty function body
21 } // end function nothing1
22
23 // function that receives arguments of types
24 // char, int, float & and double &
25 int nothing2( char a, int b, float &c, double &d )
26 {
27    return 0;
28 } // end function nothing2
29
30 int main()
31 {
32    return 0; // indicates successful termination
33 } // end main
```

```
@square$qi
@square$qd
@nothing1$qifcri
@nothing2$qcirfrd
_main
```

图 6.25　改编名称以保证类型安全的连接



常见的编程错误 6.21
创建具有相同参数列表和不同返回类型的重载函数会产生编译错误。

编译器只使用参数列表来区分具有相同名字的函数。重载函数不需要具有相同个数的参数。当重载具有默认参数的函数时，程序员应该格外小心，因为这可能引起二义性。



常见的编程错误 6.22
调用具有默认实参的函数时省略实参其形式可能会与调用另一个重载的函数一样，这会产生编译错误。例如，在程序中有两个函数，函数名相同，但其中一个函数没有参数，而另一个函数带有所有默认实参，如果在一个不传递实参的调用中使用函数名，就会导致编译错误。编译器不知道选择哪一个版本的函数。

重载的运算符

在第 11 章中，我们将讨论如何重载运算符，从而定义其如何操作于用户自定义数据类型的对象（事实上，到目前为止，我们一直在使用许多重载的运算符，包括流插入运算符<<和流提取运算符>>，这两个运算符都被重载以便能够显示所有的基本类型数据。我们会在第 11 章中更详细地讲述关于重载<<和>>来处理用户自定义类型的对象）。6.18 节介绍自动生成在不同数据类型上执行相同任务的重载函数的函数模板。

6.18 函数模板

重载函数通常用于执行相似的操作，这些操作涉及到应用于不同数据类型上的不同程序逻辑。如果对于每种数据类型程序逻辑和操作都是相同的，那么使用函数模板可以更为简便地对函数进行重载。程序员需要编写单个函数模板定义。假设在这个模板函数调用中提供了实参类型，C++会自动生成独立的函数模板“特化”（specialization）来恰如其分地处理调用的每种类型。这样，定义一个函数模板实质上就定义了一整套重载函数。

图 6.26 包含了 maximum 函数的函数模板的定义（第 4~18 行），maximum 函数用于确定三个数值中的最大值。所有的函数模板定义都以 template 关键字开头（第 4 行），后跟用一对尖括号（<和>）括起的该函数模板的模板参数列表。模板参数列表中的每个参数（常常称做形式类型参数）由关键字 typename 或者关键字 class（它们是同义词）开头。形式类型参数是基本类型或者用户自定义类型的占位符。这些占位符用于指定函数参数的类型（第 5 行）、指定函数的返回类型（第 5 行）和在函数定义体内声明变量（第 7 行）。函数模板的定义与其他函数的定义一样，只是使用形式类型参数作为实际数据类型的占位符。

```
1  // Fig. 6.26: maximum.h
2  // Definition of function template maximum.
3
4  template < class T >  // or template< typename T >
5  T maximum( T value1, T value2, T value3 )
6  {
7     T maximumValue = value1; // assume value1 is maximum
8
9     // determine whether value2 is greater than maximumValue
10    if ( value2 > maximumValue )
11       maximumValue = value2;
12
13    // determine whether value3 is greater than maximumValue
14    if ( value3 > maximumValue )
15       maximumValue = value3;
16
17    return maximumValue;
18 } // end function template maximum
```

图 6.26 函数模板 maximum 头文件

图 6.26 中的函数模板声明了单个形式类型参数 T（第 4 行）作为占位符，为 maximum 函数测试数据类型做准备。对于一个特定的模板定义，类型参数的名字在模板参数列表中必须是惟一的。当编译器在程序源代码中检测到 maximum 调用时，传递给 maximum 的数据类型代替整个模板定义中的 T，并且为了确定三个给定数据类型值的最大值，C++ 会创建一个完整的函数，然后编译新创建的函数。因此，模板是代码生成的一种方法。



常见的编程错误 6.23

在函数模板的每个形式类型参数前不加关键字 class 或者 typename（例如，写成< class S, T >而不是< class S, class T >）将导致语法错误。

图 6.27 使用函数模板 maximum（第 20、30 和 40 行）来分别确定三个 int 值、三个 double 值和三个 char 值的最大值。

```
1  // Fig. 6.27: fig06_27.cpp
2  // Function template maximum test program.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include "maximum.h" // include definition of function template maximum
9
10 int main()
11 {
12    // demonstrate maximum with int values
13    int int1, int2, int3;
14
15    cout << "Input three integer values: ";
16    cin >> int1 >> int2 >> int3;
17
18    // invoke int version of maximum
19    cout << "The maximum integer value is: "
20       << maximum( int1, int2, int3 );
21
22    // demonstrate maximum with double values
23    double double1, double2, double3;
24
25    cout << "\n\nInput three double values: ";
26    cin >> double1 >> double2 >> double3;
27
28    // invoke double version of maximum
29    cout << "The maximum double value is: "
30       << maximum( double1, double2, double3 );
31
32    // demonstrate maximum with char values
33    char char1, char2, char3;
34
35    cout << "\n\nInput three characters: ";
36    cin >> char1 >> char2 >> char3;
37
38    // invoke char version of maximum
39    cout << "The maximum character value is: "
40       << maximum( char1, char2, char3 ) << endl;
41    return 0; // indicates successful termination
42 } // end main
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

图 6.27　演示函数模板 maximum

对于大于或者等于 0 的整数 number，其阶乘可以通过使用下面的 for 语句迭代地（非递归地）进行计算：

```
factorial = 1;

for ( int counter = number; counter >= 1; counter-- )
   factorial *= counter;
```

通过观察下面的关系：

$n! = n \cdot (n-1)!$

我们可以总结出阶乘函数的递归定义。

例如，很明显 5!等于 5 * 4!，如下所示：

$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$

$5! = 5 \cdot (4!)$

5!的求值过程如图 6.28 所示。图 6.28(a)展示了一系列连续的递归调用如何进行，直到 1!的结果为 1，这时递归终止。图 6.28(b)显示了每次递归调用向其调用者返回的值，直到计算并返回最后的值。

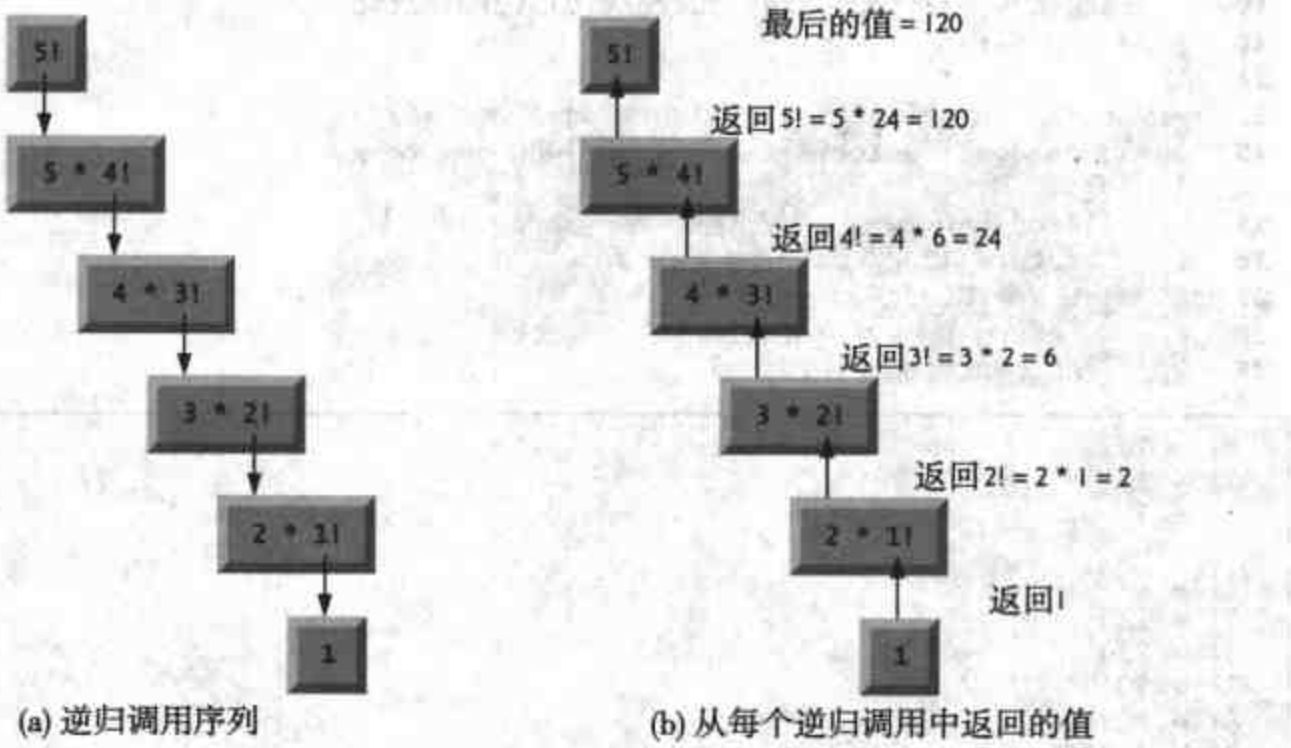


图 6.28　5!的递归求值过程

图 6.29 中的程序采用递归计算并打印 0 ~ 10 之间整数的阶乘（后面会解释选择数据类型 unsigned long 的原因）。递归函数 factorial（第 23 ~ 29 行）首先确定终止条件 number<=1（第 25 行）是否为真。如果 number 确实小于或等于 1，那么 factorial 函数返回 1（第 26 行），就不再需要更多的递归并且函数终止。如果 number 大于 1，第 28 行将问题表示为 number 与计算 number–1 阶乘的对 factorial 的递归调用的乘积。注意 factorial(number – 1)是比原来的计算 factorial(number)更简单的问题。

factorial 函数声明为接受 unsigned long 类型的参数，并返回类型为 unsigned long 的结果。unsigned long 是 unsigned long int 的缩写。C++ 标准文档规定 unsigned long int 类型的变量至少需要 4 个字节（32 位）存储；因此，它的取值范围是在 0 到 4 294 967 295 之间（数据类型 long int 也要至少 4 个字节保存，取值的范围在 –2 147 483 648 到 2 147 483 647 之间）。如图 6.29 所示，阶乘的值很快变得很大。我们选择数据类型 unsigned long 以便程序可以在具有小整数（例如 x 字节）的计算机上计算大于 7!的阶乘。遗憾的是，factorial 函数很快就能产生很大的值，使得甚至在超出 unsigned long 变量的长度之前，unsigned long 也不能用于计算多个阶乘的值。

习题中要求使用数据类型为 double 的变量来计算大数的阶乘。这指出了大多数程序设计语言的弱点，换言之，编程语言不能很容易地扩展以处理各种应用程序的专门需求。在较深入地讨论面向对象编程时，我们会看到 C++ 是一种可扩展语言，允许我们创建能够表示任意想要的大整数的类。在流行的类库中已经提供了这种类[①]，我们将在习题 9.14 和习题 11.5 中处理类似的类。

```
// Fig. 6.29: fig06_29.cpp
// Testing the recursive factorial function.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

unsigned long factorial( unsigned long ); // function prototype

int main()
{
   // calculate the factorials of 0 through 10
   for ( int counter = 0; counter <= 10; counter++ )
      cout << setw( 2 ) << counter << "! = " << factorial( counter )
         << endl;

   return 0; // indicates successful termination
} // end main

// recursive definition of function factorial
unsigned long factorial( unsigned long number )
{
   if ( number <= 1 ) // test for base case
      return 1; // base cases: 0! = 1 and 1! = 1
   else // recursion step
      return number * factorial( number - 1 );
} // end function factorial
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

图 6.29　演示 factorial 函数



常见的编程错误 6.24

省略基本情况，或者将递归步骤写得不正确以至于它不能归结到基本情况，都会引起“无限”递归，最终耗尽内存。这和循环（非递归）解决方案中的无限循环问题类似。

6.20　递归应用示例：Fibonacci 数列

Fibonacci 数列

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

从 0，1 开始，后面的每个 Fibonacci 数是其前面两个 Fibonacci 数之和。

① 这些类可以在shoup.net/ntl, cliodhna.cop.uop.edu/~hetrick/c-sources.html和www.trumphurst.com/cpplibs/datapage.phtml?category='intro' 中找到。

Fibonacci数列存在于自然界中，尤其是它描述了一种螺旋形状。相邻的Fibonacci数之比大约是1.618。这个数也常常出现在自然世界中，称为黄金比例或者黄金分割（golden mean）。人们发现黄金分割在美学上最令人满意。建筑师在设计窗户、房间和建筑物时，常使它们的长宽比满足黄金分割比例。明信片的长宽比通常也是满足黄金比例的。

Fibonacci数列可以递归地定义如下：

fibonacci(0) = 0

fibonacci(1) = 1

fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

图6.30中的程序用fibonacci函数递归计算第n个Fibonacci数。注意，虽然Fibonacci数列增长速度比阶乘慢一些，但是Fibonacci数也呈迅速变大的趋势。因此，在fibonacci函数中我们选择unsigned long数据类型作为函数参数和返回值的类型。图6.30中给出的程序的执行情况，显示了几个数对应的Fibonacci值。

```
1  // Fig. 6.30: fig06_30.cpp
2  // Testing the recursive fibonacci function.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  unsigned long fibonacci( unsigned long ); // function prototype
9
10 int main()
11 {
12    // calculate the fibonacci values of 0 through 10
13    for ( int counter = 0; counter <= 10; counter++ )
14       cout << "fibonacci( " << counter << " ) = "
15          << fibonacci( counter ) << endl;
16
17    // display higher fibonacci values
18    cout << "fibonacci( 20 ) = " << fibonacci( 20 ) << endl;
19    cout << "fibonacci( 30 ) = " << fibonacci( 30 ) << endl;
20    cout << "fibonacci( 35 ) = " << fibonacci( 35 ) << endl;
21    return 0; // indicates successful termination
22 } // end main
23
24 // recursive method fibonacci
25 unsigned long fibonacci( unsigned long number )
26 {
27    if ( ( number == 0 ) || ( number == 1 ) ) // base cases
28       return number;
29    else // recursion step
30       return fibonacci( number - 1 ) + fibonacci( number - 2 );
31 } // end function fibonacci
```

```
fibonacci( 0 ) = 0
fibonacci( 1 ) = 1
fibonacci( 2 ) = 1
fibonacci( 3 ) = 2
fibonacci( 4 ) = 3
fibonacci( 5 ) = 5
fibonacci( 6 ) = 8
fibonacci( 7 ) = 13
fibonacci( 8 ) = 21
fibonacci( 9 ) = 34
fibonacci( 10 ) = 55
fibonacci( 20 ) = 6765
fibonacci( 30 ) = 832040
fibonacci( 35 ) = 9227465
```

图 6.30　演示 fibonacci 函数

这个应用程序从一条 for 语句开始，计算和显示了 0 ~ 10 之间整数对应的 Fibonacci 值。此 for 语句之后紧跟着三个函数调用，用于计算整数 20、30 和 35（第 18 ~ 20 行）对应的 Fibonacci 值。由 main 函数调用 fibonacci（第 15、18、19 和 20 行）不是递归调用，但是第 30 行的调用是递归的。程序每次调用 fibonacci 时（第 25 ~ 31 行），函数立即检测基本情况以确定 number 是否等于 0 或 1（第 27 行）。如果是，在第 28 行返回 number。有趣的是，如果 number 大于 1，递归步骤（第 30 行）生成两个递归调用，每个都针对比原来的 fibonacci 调用小一些的问题。图 6.31 显示的是 fibonacci 函数如何计算 fibonacci(3)的值。

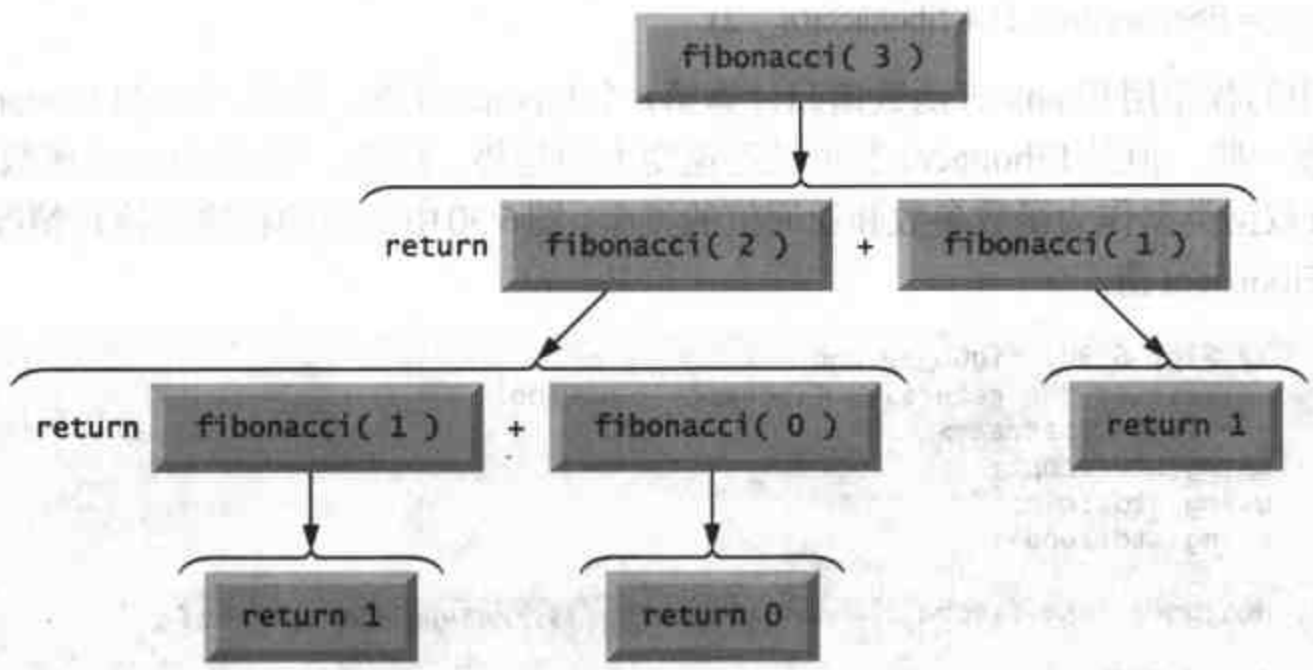


图 6.31 fibonacci 函数的递归调用序列

图 6.31 引出了关于 C++ 编译器对运算符的操作数进行求值的顺序的一些有趣问题。这不同于运算符应用于其操作数的顺序问题，换句话说，后者是由运算符的优先级和结合律的规则所决定的顺序问题。图 6.31 显示求 fibonacci(3)的值引起了两个递归调用，即 fibonacci(2)和 fibonacci(1)。但是这些调用的执行顺序又如何呢？

大部分程序员简单地假定操作数是按照从左到右的顺序求值的。C++语言没有指定大多数运算符（包括 + 在内）其操作数的求值顺序。因此，程序员不应该对这些调用的执行顺序做出假定。事实上，这些调用可能先执行 fibonacci(2)，然后执行 fibonacci(1)，或者也可能反过来执行：先执行 fibonacci(1)，再执行 fibonacci(2)。在本程序以及大多数程序中，最后的结果都是相同的。但是，在某些程序中，操作数的求值顺序具有副作用（改变了数据的值），会影响表达式的最终结果。

C++ 语言只指定了四种运算符操作数的求值顺序，这四种运算符分别是 &&、||、,和?:。前三个是二元运算符，它们的两个操作数是按从左到右的顺序进行求值的。最后一个运算符是C++中惟一的三元运算符。其最左边的操作数总是先被求值；如果最左边的操作数的求值结果为非零值（真），就接着求中间的操作数的值，最后的操作数被忽略；如果最左边的操作数的求值结果为0（假），接着就求第三个操作数的值，中间的操作数被忽略。



常见的编程错误 6.25

编写的程序，如果依赖于非 &&、||、,和?:运算符操作数的求值顺序，可能会导致逻辑错误。



可移植性提示 6.3

依赖于非 &&、||、,和?:运算符操作数的求值顺序的程序在使用不同编译器时会有不同的系统表现。

应该谨慎地使用递归程序，就像我们在这里用来生成 Fibonacci 数的情况。fibonacci 函数中的每级递归都使函数调用的次数加倍，换言之，计算第 n 个 Fibonacci 数的递归调用的次数是 2^n。这

样很快就会失去控制。仅仅计算第 20 个 Fibonacci 数就需要 2^{20} 或者说上百万次调用，而计算第 30 个 Fibonacci 数就需要 2^{30} 或上十亿次调用，依次类推。计算机科学家把这种增长级别称为指数级复杂度。即使是世界上最强大的计算机恐怕也无法胜任这种复杂度的问题。关于复杂度的问题，特别是指数级复杂度的问题，将在更高级的计算机科学课程“算法”中详细讨论。



性能提示 6.8
应该避免导致指数级调用“爆炸”的 Fibonacci 式的递归程序。

6.21　递归与迭代

在前面的两节中，我们研究了两个可以很容易地用递归或者迭代实现的函数。本节比较这两种方法并讨论为什么程序员会在某些特定情况下选择一种方法而不是另一种方法。

迭代和递归都是基于控制语句的：迭代使用循环结构；递归使用选择结构。迭代和递归都涉及到循环：迭代显式地使用循环结构；递归通过重复的函数调用实现循环。迭代和递归均包括终止条件测试：迭代在循环继续条件失败时终止；递归在达到基本情况时终止。计数器控制的循环的迭代和递归都是逐步达到终止的：迭代修改计数器直到计算器的值使循环条件不满足；递归产生比原来的问题简单的版本直到达到基本情况。迭代和递归都可能无限进行：如果循环继续测试一直不变成假，则迭代会发生无限循环；如果递归步骤不能通过递归调用归结到基本情况，就会导致无限递归。

为了说明迭代和递归间的差异，我们来看一个用迭代的方法处理阶乘的例子（见图 6.32）。注意迭代使用了循环语句（见图 6.32 中的第 28 ~ 29 行）而不是递归方法的选择语句（见图 6.29 中的第 24 ~ 27 行）。两种方法都需要进行终止测试。在递归方法中，第 24 行测试了基本情况。在迭代方法中，第 28 行测试了循环继续条件——如果测试失败，则循环终止。最后，注意，代替生成原来问题的简单版本，迭代方法使用一个计数器，并且在循环过程中修改它的值，直到循环继续条件变成假。

```
1  // Fig. 6.32: fig06_32.cpp
2  // Testing the iterative factorial function.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 unsigned long factorial( unsigned long ); // function prototype
11
12 int main()
13 {
14    // calculate the factorials of 0 through 10
15    for ( int counter = 0; counter <= 10; counter++ )
16       cout << setw( 2 ) << counter << "! = " << factorial( counter )
17          << endl;
18
19    return 0;
20 } // end main
21
22 // iterative function factorial
23 unsigned long factorial( unsigned long number )
24 {
25    unsigned long result = 1;
26
```

图 6.32　用迭代方法解决阶乘问题

```
   // iterative declaration of function factorial
   for ( unsigned long i = number; i >= 1; i-- )
      result *= i;

   return result;
} // end function factorial
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

图 6.32 用迭代方法解决阶乘问题（续）

递归有许多不足之处。它不断地进行函数调用，必然会增加很多开销。这样不仅消耗处理器的时间，还会消耗内存空间。每个递归调用都会创建函数的一份副本（实际上只是函数中的变量），这也会占用相当可观的内存。而迭代通常发生在一个函数内，因此没有重复的函数调用的开销和额外的内存分配。那么为什么还要选择递归呢？



软件工程知识 6.18

任何可以用递归解决的问题都可以用迭代（非递归）解决。如果使用递归方法能够更自然地反映问题，并且能够使程序易于理解和调试，那么我们就选择递归方法而不是迭代方法。选择递归方法的另一个原因是迭代方法实现起来比较晦涩。



性能提示 6.9

在要求性能的情况下应避免使用递归。递归调用会消耗额外的时间和内存。



常见的编程错误 6.26

如果一个非递归函数不经意地直接或者间接（通过另外一个函数）调用了自身，就会产生逻辑错误。

与我们的做法不同，大多数程序设计的教科书都会在较后的章节才介绍递归。我们认为递归是有意义的且复杂的问题，最好尽早介绍，然后在后面的章节中再介绍一些例子。图 6.33 总结了本书中的递归示例和习题。

书中章节	递归示例和习题
第 6 章	
6.19 节，图 6.29	阶乘函数
6.19 节，图 6.30	Fibonacci 函数
习题 6.7	两个整数之和
习题 6.40	求一个整数的整数幂
习题 6.42	Hanoi 塔问题
习题 6.44	可视化递归
习题 6.45	最大公约数
习题 6.50，习题 6.51	神秘的习题“这个程序做什么？”

图 6.33 本书中递归例子和习题总结

k) 存储类别说明符有 mutable、___、___、___和___。

l) 除非另外指定，否则在语句块中或者函数参数列表中声明的变量默认情况下是___存储类别的。

m) ___存储类别说明符建议编译器将变量保存在计算机的寄存器中。

n) 在所有的语句块和函数外声明的变量是___变量。

o) 为了在函数调用间保留函数中局部变量的值，局部变量必须用___存储类别说明符声明。

p) 标识符的六个可能的作用域是___、___、___、___、___和___。

q) 直接或者间接地（即通过另外一个函数）调用自身的函数称为___函数。

r) 递归函数一般有两个组成部分：一部分提供了通过测试___情况来终止递归的方法，另一部分将问题表示为对稍微简单点问题的递归调用。

s) 在C++中，可能有多个函数具有相同的名字，但是它们的参数类型或者参数个数不同。这称为函数___。

t) ___可以在当前局部变量作用域中访问同名的全局变量。

u) ___限定词用于声明只读变量。

v) 函数___使一个函数可以定义成能在许多不同的数据类型上执行一项任务。

6.2 对于图6.34所示的程序，说出下面每个元素的作用域（函数作用域、文件作用域、语句块作用域或者函数原型作用域）：

a) main 中的 x 变量。

b) cube 中的 y 变量。

c) cube 函数。

d) main 函数。

e) cube 的函数原型。

f) 在 cube 函数原型中的 y 标识符。

```
1  // Exercise 6.2: Ex06_02.cpp
2  #include <iostream>
3  using std::cout;
4  using std::endl;
5
6  int cube( int y ); // function prototype
7
8  int main()
9  {
10    int x;
11
12    for ( x = 1; x <= 10; x++ ) // loop 10 times
13       cout << cube( x ) << endl; // calculate cube of x and output results
14
15    return 0; // indicates successful termination
16 } // end main
17
18 // definition of function cube
19 int cube( int y )
20 {
21    return y * y * y;
22 } // end function cube
```

图 6.34　自测题 6.2 的程序

6.3 编写一个程序，测试图6.2中给出的数学库函数调用的例子，看看是否真地得到图示的结果。

6.4 为下面的每个函数写出函数头部：

```
   int result;
   cout << "Enter three integers: ";
   cin >> a >> b >> c;
   result = a * b * c;
   cout << "Result is " << result;
   return result;
}
```

6.8　为什么有的函数原型中有诸如 double& 这样的参数类型声明？

6.9　（判断对错）C++ 函数调用中的所有实参都是按值传递的。

6.10　编写一个完整的程序，提示用户输入球的半径，计算和打印球的体积。使用内联函数 sphereVolume，返回表达式(4.0 / 3.0)* 3.14159 * pow(radius, 3)的结果。

自测题答案

6.1　a) 函数，类。b) 函数调用。c) 局部变量。d) return。e) void。f) 作用域。g) return;，return 表达式; 或者到达函数结束的右花括号。h) 函数原型。i) rand。j) srand。k) auto, register, extern, static。l) auto。m) register。n) 全局。o) static。p) 函数作用域，文件作用域，语句块作用域，函数原型作用域，类作用域和命名空间作用域。q) 递归。r) 基本。s) 重载。t) 一元作用域分辨运算符（::）。u) const。v) 模板。

6.2　a) 语句块作用域。b) 语句块作用域。c) 文件作用域。d) 文件作用域。e) 文件作用域。f) 函数原型作用域。

6.3　参见下面的程序：

```
1  // Exercise 6.3: Ex06_03.cpp
2  // Testing the math library functions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 #include <cmath>
12 using namespace std;
13
14 int main()
15 {
16    cout << fixed << setprecision( 1 );
17
18    cout << "sqrt(" << 900.0 << ") = " << sqrt( 900.0 )
19       << "\nsqrt(" << 9.0 << ") = " << sqrt( 9.0 );
20    cout << "\nexp(" << 1.0 << ") = " << setprecision( 6 )
21       << exp( 1.0 ) << "\nexp(" << setprecision( 1 ) << 2.0
22       << ") = " << setprecision( 6 ) << exp( 2.0 );
23    cout << "\nlog(" << 2.718282 << ") = " << setprecision( 1 )
24       << log( 2.718282 )
25       << "\nlog(" << setprecision( 6 ) << 7.389056 << ") = "
26       << setprecision( 1 ) << log( 7.389056 );
27    cout << "\nlog10(" << 1.0 << ") = " << log10( 1.0 )
28       << "\nlog10(" << 10.0 << ") = " << log10( 10.0 )
29       << "\nlog10(" << 100.0 << ") = " << log10( 100.0 );
30    cout << "\nfabs(" << 13.5 << ") = " << fabs( 13.5 )
31       << "\nfabs(" << 0.0 << ") = " << fabs( 0.0 )
32       << "\nfabs(" << -13.5 << ") = " << fabs( -13.5 );
33    cout << "\nceil(" << 9.2 << ") = " << ceil( 9.2 )
```

```
      << "\nceil(" << -9.8 << ") = " << ceil( -9.8 );
   cout << "\nfloor(" << 9.2 << ") = " << floor( 9.2 )
      << "\nfloor(" << -9.8 << ") = " << floor( -9.8 );
   cout << "\npow(" << 2.0 << ", " << 7.0 << ") = "
      << pow( 2.0, 7.0 ) << "\npow(" << 9.0 << ", "
      << 0.5 << ") = " << pow( 9.0, 0.5 );
   cout << setprecision(3) << "\nfmod("
      << 13.675 << ", " << 2.333 << ") = "
      << fmod( 13.675, 2.333 ) << setprecision( 1 );
   cout << "\nsin(" << 0.0 << ") = " << sin( 0.0 );
   cout << "\ncos(" << 0.0 << ") = " << cos( 0.0 );
   cout << "\ntan(" << 0.0 << ") = " << tan( 0.0 ) << endl;
   return 0; // indicates successful termination
} // end main
```

```
sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0
fabs(13.5) = 13.5
fabs(0.0) = 0.0
fabs(-13.5) = 13.5
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
fmod(13.675, 2.333) = 2.010
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0
```

6.4 a) double hypotenuse(double side1, double side2)

b) int smallest(int x, int y, int z)

c) void instructions(void)//in C++(void)can be written()

d) double intToDouble(int number)

6.5 a) double hypotenuse(double, double);

b) int smallest(int, int, int);

c) void instructions(void); //in C++(void)can be written()

d) double intToDouble(int);

6.6 a) register int count = 0;

b) static double lastVal;

6.7 a) 错误：函数 h 定义在函数 g 中。
改正：把函数 h 的定义移到函数 g 的外面。

b) 错误：函数应该返回一个整数，但却没有。
改正：删除变量 result 并把语句 return x + y;放到函数中。

c) 错误：n + sum(n – 1)的结果没有返回； sum 返回了一个不正确的结果。
改正：把 else 子句中的语句改写成 return n + sum(n – 1);。

d) 错误：括住参数列表的右括号后出现了分号，并且在函数定义中重定义了参数 a。

改正：删除参数列表右括号后的分号，并删除声明 float a;。

e) 错误：在不该返回值的函数中返回了值。

改正：去掉 return 语句。

6.8　这创建了 double 引用类型的引用参数，使函数能够修改调用函数中的原始变量。

6.9　错。C++ 可以使用引用参数（和指针，我们将在第 8 章中讨论）来进行参数的按引用传递。

6.10　参见下面的程序：

```
1  // Exercise 6.10 Solution: Ex06_10.cpp
2  // Inline function that calculates the volume of a sphere.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  #include <cmath>
9  using std::pow;
10
11 const double PI = 3.14159; // define global constant PI
12
13 // calculates volume of a sphere
14 inline double sphereVolume( const double radius )
15 {
16    return 4.0 / 3.0 * PI * pow( radius, 3 );
17 } // end inline function sphereVolume
18
19 int main()
20 {
21    double radiusValue;
22
23    // prompt user for radius
24    cout << "Enter the length of the radius of your sphere: ";
25    cin >> radiusValue; // input radius
26
27    // use radiusValue to calculate volume of sphere and display result
28    cout << "Volume of sphere with radius " << radiusValue
29       << " is " << sphereVolume( radiusValue ) << endl;
30    return 0; // indicates successful termination
31 } // end main
```

习题

6.11　在下面的每条语句执行后显示 x 的值：

a) x = fabs(7.5)

b) x = floor(7.5)

c) x = fabs(0.0)

d) x = ceil(0.0)

e) x = fabs(-6.4)

f) x = ceil(-6.4)

g) x = ceil(-fabs(-8 + floor(-5.5)))

6.12　停车场三小时内的最少收费是 2.00 美元。超过三小时，每增加 1 小时需要收取 0.50 美元的附加费用。24 小时之内的最多收费是 10 美元。假设没有车子一次停车时间超过 24 小时。编写一个程序，计算并显示昨天的三个客户各自的停车费用。要求应该输入每个客

b) 3, 5, 7, 9, 11.

c) 6, 10, 14, 18, 22.

6.18 编写一个函数 integerPower(base, exponent)，它返回下面的值：

`base` *exponent*

例如，integerPower(3, 4)= 3 * 3 * 3 * 3。假设 exponent 是正的非零整数并且 base 是整数。函数 integerPower 应该使用 for 或者 while 来控制计算。不要使用任何数学库函数。

6.19（直角三角形之斜边）定义一个函数 hypotenuse，用于计算在已知直角三角形的两条直角边边长时其斜边的边长。在程序中使用这个函数，计算下面给出的每个三角形的斜边边长。该函数有两个 double 类型的参数，并返回 double 类型的斜边边长。

Triangle	Side 1	Side 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

6.20 编写一个函数 multiple，用于确定一对整数中第二个整数是否是第一个整数的倍数。函数应该需要两个整数参数，并且如果第二个整数是第一个整数的倍数的话，就返回 true，否则返回 false。在程序中使用这个函数判断输入的一系列整数对。

6.21 编写一个程序，输入一系列整数，并将每个整数分别传递给函数 even，该函数利用取模运算符来确定一个整数是否为偶数。函数应该有一个整数参数，并且如果整数为偶数，则返回 true，否则返回 false。

6.22 编写一个函数，在屏幕的左边空白处显示一个由星号组成的实体正方形，边长由整数参数 side 指定。例如，如果 side 为 4，函数就会显示如下结果：

```
****
****
****
****
```

6.23 修改习题 6.22 创建的函数，通过给定的字符参数 fillCharacter，就可以得到由该字符参数中的字符组成的正方形。因此，如果 side 等于 5 并且 fillCharacter 是 #，那么这个函数打印出如下结果：

```
#####
#####
#####
#####
#####
```

6.24 利用类似于习题 6.22 和习题 6.23 中的开发技巧，编写一个可以绘制很多形状的程序。

6.25 编写程序段，分别实现如下任务：

a) 计算整数 a 除以整数 b 得到的商的整数部分。

b) 计算整数 a 除以整数 b 得到的整数余数。

c) 利用在 a)和 b)中开发的程序片段编写一个函数，该函数输入 1 到 32767 之间的整数，打印出这个数的数字序列，该数字序列中两个数字间用两个空格分开。例如，整数 4562 应打印为：

6.40 编写一个递归函数 power(base,exponent)，它在调用时返回

$base^{exponent}$

例如，power(3, 4)= 3 * 3 * 3 * 3。假设 exponent 是大于或等于 1 的整数。提示：递归步骤将利用如下关系：

$base^{exponent} = base \cdot base^{exponent - 1}$

exponent 等于 1 是递归终止条件，因为

$base^{1} = base$

6.41（Fibonacci 数列）Fibonacci 数列：

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

从 0 和 1 开始，后续的数具有这样的性质：当前的数是其前面两个数之和。(a)编写一个非递归的fibonacci(n)函数，计算第n个Fibonacci数。(b)确定你机器上可以打印的最大int类型的Fibonacci数。用 double 代替 int 修改(a)部分的程序，计算并返回 Fibonacci 数，并使用修改的程序重复执行(b)部分的任务。

6.42（汉诺塔问题）在本章中，读者学习函数既可以用递归很容易地实现，也可以用迭代很容易地实现。在这道习题中，我们提出的问题若用递归来解决，则尽显递归之优雅；若用迭代来实现，恐怕没那么容易。

汉诺塔问题是每个新一代的计算机科学家必须掌握的最著名的经典问题之一。传说在遥远的东方有一座庙，僧侣们尝试把一叠金盘从一根桩上移到另一个根桩上（见图 6.35）。起初有64个金盘串在一个桩上，从下到上尺寸逐步缩小。僧侣们尝试着按照一次只能移动一个金盘并且大的金盘永远不能放在小的金盘上面的规定，来将这叠金盘移动到另外一个桩上。总共有三个桩，一个用于暂放金盘。按照推测，僧侣们完成他们的工作之时，正是地球毁灭之日。若真是这样，我们可不愿意助他们一臂之力了。

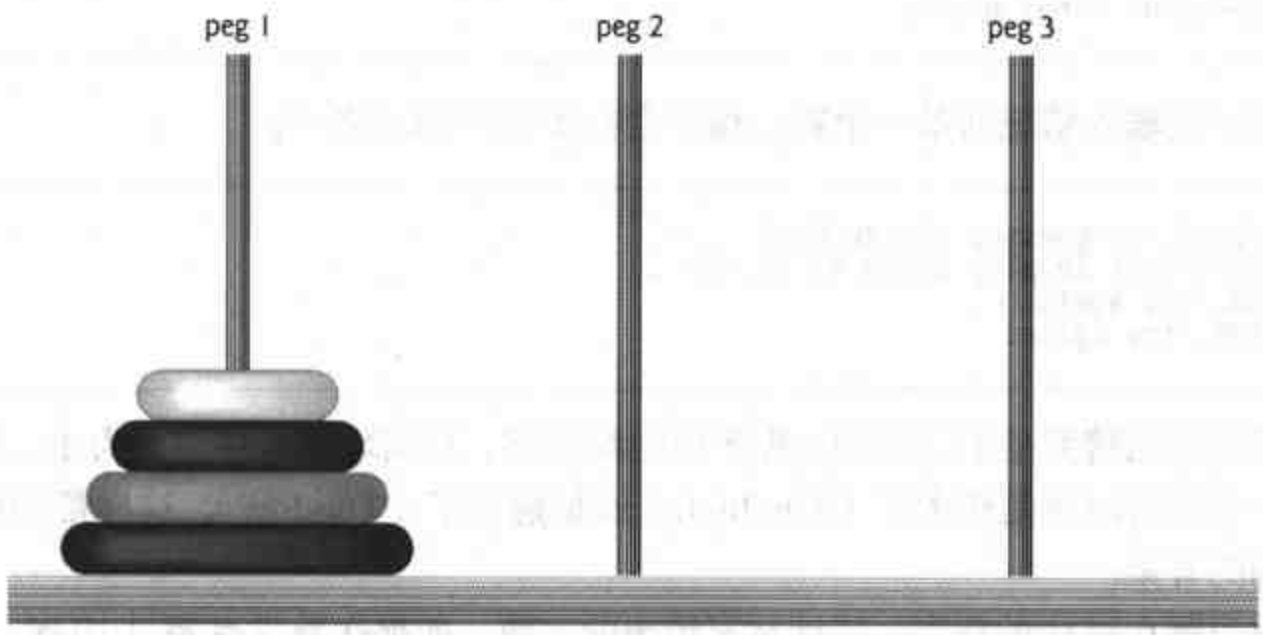


图 6.35　有四个盘子的汉诺塔问题

让我们假设僧侣们想把盘子从桩1移到桩3。我们希望开发一个算法，显示僧侣从桩到桩移动盘子的序列。

如果我们用传统的方法来处理这个问题的话，会很快发现我们陷入到这堆盘子的管理之中而无法自拔。这个问题很棘手，似乎没有什么希望解决。然而，用递归的方法来处理

a) 修改程序，允许用户输入能力等级。等级 1 表示在问题中只能使用一位数，等级 2 表示最大只能使用 2 位数，等等。

b) 修改程序，允许用户选择他或者她希望学习的算术问题的类型。选项 1 表示只是加法问题，2 表示只是减法问题，3 表示只是乘法问题，4 表示只是除法问题，5 表示所有类型随机混合的问题。

6.48 编写一个函数 distance，计算两点(*x1*, *y1*)和(*x2*, *y2*)之间的距离。所有的数和返回值都应该是 double 类型的。

6.49 下面的程序错在哪里？

```
1  // Exercise 6.49: ex06_49.cpp
2  // What is wrong with this program?
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6
7  int main()
8  {
9     int c;
10
11    if ( ( c = cin.get() ) != EOF )
12    {
13       main();
14       cout << c;
15    } // end if
16
17    return 0; // indicates successful termination
18 } // end main
```

6.50 下面的程序做什么？

```
1  // Exercise 6.50: ex06_50.cpp
2  // What does this program do?
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  int mystery( int, int ); // function prototype
9
10 int main()
11 {
12    int x, y;
13
14    cout << "Enter two integers: ";
15    cin >> x >> y;
16    cout << "The result is " << mystery( x, y ) << endl;
17
18    return 0; // indicates successful termination
19 } // end main
20
21 // Parameter b must be a positive integer to prevent infinite recursion
22 int mystery( int a, int b )
23 {
24    if ( b == 1 ) // base case
25       return a;
26    else // recursion step
27       return a + mystery( a, b - 1 );
28 } // end function mystery
```



第7章　数组与C++标准库类模板vector

在把它们填到表中之前，即可动手记下来，并在书中加以注释。

——Isaiah

过犹不及。

——Confucius

从起点开始，……一直到抵达终点，再停下来。

——Lewis Carroll

学习目标

在本章中将学习：

- 利用数组数据结构表示一组相关的数据项
- 利用数组存储、排序和查找序列与表中的数值
- 声明数组、初始化数组、引用数组中的单个元素
- 将数组传递给函数
- 基本的查找和排序方法
- 声明和使用多维数组
- 使用C++标准库类模板vector

提纲

7.1　简介
7.2　数组
7.3　声明数组
7.4　使用数组的范例
7.5　将数组传递给函数
7.6　实例研究：利用数组存放成绩的类GradeBook
7.7　采用线性查找法查找数组
7.8　采用插值排序法排序数组
7.9　多维数组
7.10　实例研究：利用二维数组的类GradeBook
7.11　C++标准库类模板vector的介绍
7.12　本章小结

摘要 | 术语 | 自测题 | 自测题答案 | 习题 | 递归习题 | vector习题

7.1　简介

本章介绍数据结构这一重要主题。数据结构是相关数据项的集合。数组是由相同类型的相关数据项组成的数据结构。我们已经在第3章中学习了类。在第9章，我们将讨论结构体（structure）的概念。结构体和类都可以拥有不同类型的相关数据项。数组、结构体和类在程序执行期间大小保持不变，因此它们是“静态”的实体（当然，它们可以是自动存储类别的。在进入定义它们的语句块时被创建，在退出时被销毁）。

在讨论了如何声明、创建和初始化数组之后，本章给出一系列实际的例子，说明几个常见的数组操作。然后，我们解释如何用字符数组来表示字符串（到现在为止是用string对象表示的）。我们给出一个查找数组中特定元素的例子。本章还介绍了最重要的计算应用之一——排序（即把数据按照特定的次序排列）。本章中的两节增强了第3到6章中GradeBook类的实例研究。具体来说，我们利用数组，使该类可以在内存中保存一组成绩，可以根据一个学期的多次考试分析学生的成绩——这两个功能是GradeBook类以前的版本所没有的。这些例子和本章的其他例子，演示了程序员可以利用数组组织、操作数据的方法。

我们在本章中使用的数组大多是C风格的基于指针的数组（我们将在第8章学习指针）。在本章的最后一节，我们会讲到作为成熟对象的数组，即所谓的vector。我们将会发现，这些基于对象的数组比在本章前面讨论的C风格的基于指针的数组更安全、更通用。

7.2 数组

数组是一组具有相同类型的、连续的内存区域。要引用数组中的一个特定区域或元素，则通过指定数组名和该特定元素在数组中的位置编号来完成。

图7.1展示了一个名为c的整数数组。这个数组包含12个元素。通过给出后接包含元素位置编号的方括号（[]）的数组名，程序可以引用任何一个元素。更正式地，位置编号称为下标或者索引（它指出从数组开始处到当前位置的元素个数）。每个数组中第一个元素的下标都为0，因此有时称第一个元素为第0个元素。于是，数组c的元素依次是c[0]、c[1]、c[2]，等等。数组c的最大下标是11，比数组的元素个数12少1。数组名和其他变量名遵守同样的约定，即它们必须是标识符。

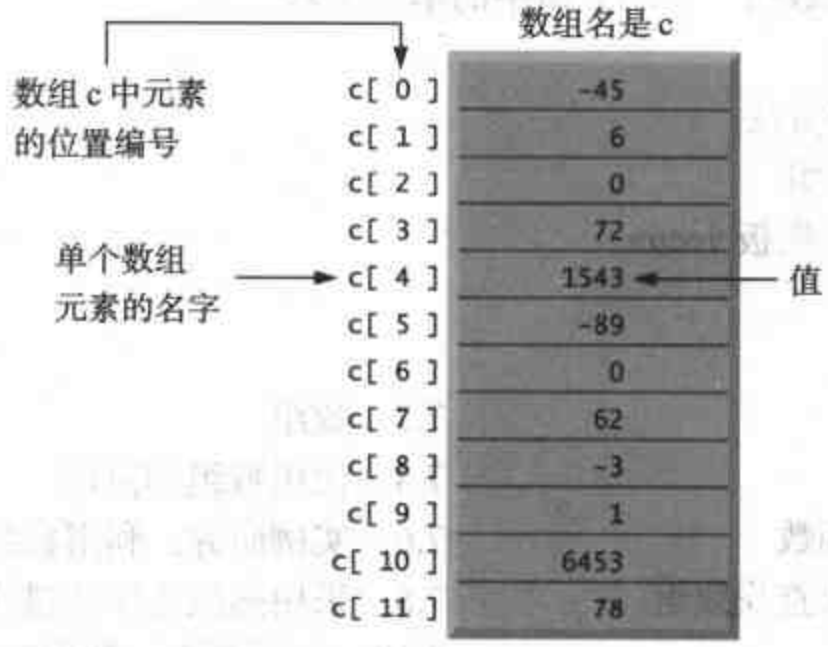


图7.1 包含12个元素的数组

下标必须是一个整数或整数表达式（使用任何整数类型）。如果程序用一个表达式作为下标，那么程序要计算这个表达式以确定下标。例如，如果我们假设变量a等于5、变量b等于6，那么如下语句：

```
c[ a + b ] += 2;
```

加2到数组元素c[11]。请注意，一个带下标的数组名是一个左值——它和其他非数组变量一样，可以在赋值语句的左边使用。

我们进一步研究图7.1中的数组c。整个数组的名字是c。数组c的12个元素分别是c[0]、c[1]、c[2]、……、c[11]。c[0]的值是-45，c[1]的值是6，c[2]的值是0，c[7]的值是62，c[11]的值是78。为了打印数组c的前三个元素的值之和，我们可编写如下语句：

符串。字符串及它们与数组的相似之处（C++从C中继承过来的一种关系），以及指针和数组之间的关系，将在第8章讨论。

7.4 使用数组的范例

这一节给出一些例子，说明如何声明数组，如何初始化数组，以及如何进行一些常见的数组操作。

声明一个数组，然后用一个循环初始化数组元素

图7.3中的程序声明了一个有10个元素的整数数组n（第12行）。第15～16行用一条for语句把数组n的所有元素都初始化为0。第一条输出语句（第18行）显示之后输出列的列标题，后面的for语句（第21～22行）以表格形式打印出该数组。记住setw指定下一个值输出时的字段宽度。

```
1  // Fig. 7.3: fig07_03.cpp
2  // Initializing an array.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    int n[ 10 ]; // n is an array of 10 integers
13
14    // initialize elements of array n to 0
15    for ( int i = 0; i < 10; i++ )
16       n[ i ] = 0; // set element at location i to 0
17
18    cout << "Element" << setw( 13 ) << "Value" << endl;
19
20    // output each array element's value
21    for ( int j = 0; j < 10; j++ )
22       cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;
23
24    return 0; // indicates successful termination
25 } // end main
```

```
Element         Value
      0             0
      1             0
      2             0
      3             0
      4             0
      5             0
      6             0
      7             0
      8             0
      9             0
```

图7.3 把数组元素初始化为0并打印数组

在声明中用初始化列表初始化一个数组

在数组的声明中，也可以用数组名后面跟一个等号和一个用逗号分隔的初始化值的列表（包含在大括号中）来初始化数组的元素。图7.4中的程序通过初始化列表用10个值初始化一个整数数组（第13行），并以表格形式打印出数组（第15~19行）。

```
1  // Fig. 7.4: fig07_04.cpp
2  // Initializing an array in a declaration.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    // use initializer list to initialize array n
13    int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
14
15    cout << "Element" << setw( 13 ) << "Value" << endl;
16
17    // output each array element's value
18    for ( int i = 0; i < 10; i++ )
19       cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
20
21    return 0; // indicates successful termination
22 } // end main
```

```
Element         Value
      0            32
      1            27
      2            64
      3            18
      4            95
      5            14
      6            90
      7            70
      8            60
      9            37
```

图7.4　在声明中初始化数组元素

如果初始化值的个数少于数组元素的个数，那么剩下的数组元素都被初始化为0。例如，下面的声明会把图7.3中数组n的元素都初始化为0：

```
int n[ 10 ] = { 0 }; // initialize elements of array n to 0
```

因为上述初始化值的个数比数组元素个数少，所以这个声明显式地把第一个元素初始化为0，同时隐式地把剩下的9个元素初始化为0。尽管static数组可以隐式地把元素初始化为0，但是自动数组并不能这样做。程序员必须用初始化列表至少初始化第一个元素为0，才能隐式地将剩下的元素初始化为0。当程序执行时，图7.3所示的初始化方法可以重复进行。

在带有初始化值列表的数组声明中，如果数组大小被省略，那么编译器通过计算初始化列表中元素的个数来确定该数组元素的个数。例如，

```
int n[] = { 1 , 2 , 3 , 4 , 5 };
```

创建一个有5个元素的数组。

如果在一个数组声明中指定了数组大小和初始化列表，那么初始值的个数必须小于或等于数组大小。下面的数组声明：

```
int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };
```

会导致一个编译错误，因为它有6个初始化值，但只有5个数组元素。



常见的编程错误 7.2
在数组的初始化列表中提供比数组元素个数多的初始值，是一个编译错误。



常见的编程错误 7.3
忘记初始化应该初始化的数组元素，是一个逻辑错误。

用常量变量指定数组大小，用计算结果设定数组元素

图7.5中的程序把一个有10个元素的数组s的元素设置为偶数2、4、6、……、20（第17~18行），并以表格形式打印这个数组（第20~24行）。这些数是由循环计数器对每次迭代时的值乘以2再加上2得到的（第18行）。

```
1  // Fig. 7.5: fig07_05.cpp
2  // Set array s to the even integers from 2 to 20.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    // constant variable can be used to specify array size
13    const int arraySize = 10;
14
15    int s[ arraySize ]; // array s has 10 elements
16
17    for ( int i = 0; i < arraySize; i++ ) // set the values
18       s[ i ] = 2 + 2 * i;
19
20    cout << "Element" << setw( 13 ) << "Value" << endl;
21
22    // output contents of array s in tabular format
23    for ( int j = 0; j < arraySize; j++ )
24       cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
25
26    return 0; // indicates successful termination
27 } // end main
```

```
Element         Value
      0             2
      1             4
      2             6
      3             8
      4            10
      5            12
      6            14
      7            16
      8            18
      9            20
```

图 7.5　生成数组元素的值

第13行使用const限定词声明一个所谓的常量变量（constant variable）arraySize，其值为10。常量变量必须在声明时用一个常量表达式来初始化，而且之后不能修改（如图7.6和图7.7所示）。常量变量也称为命名常量（named constant）或只读变量（read-only variable）。



常见的编程错误 7.4
在声明常量变量时没有给它赋值是一个编译错误。



常见的编程错误 7.5
在可执行语句中给常量变量赋值是一个编译错误。

```
1  // Fig. 7.6: fig07_06.cpp
2  // Using a properly initialized constant variable.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     const int x = 7; // initialized constant variable
10
11    cout << "The value of constant variable x is: " << x << endl;
12
13    return 0; // indicates successful termination
14 } // end main
```

```
The value of constant variable x is: 7
```

图 7.6　初始化和使用常量变量

```
1  // Fig. 7.7: fig07_07.cpp
2  // A const variable must be initialized.
3
4  int main()
5  {
6     const int x; // Error: x must be initialized
7
8     x = 7; // Error: cannot modify a const variable
9
10    return 0; // indicates successful termination
11 } // end main
```

Borland C++ command-line compiler error message:

```
Error E2304 fig07_07.cpp 6: Constant variable 'x' must be initialized
   in function main()
Error E2024 fig07_07.cpp 8: Cannot modify a const object in function main()
```

Microsoft Visual C++.NET compiler error message:

```
C:\scpphtp5_examples\ch07\fig07_07.cpp(6) : error C2734: 'x' : const object
   must be initialized if not extern
C:\scpphtp5_examples\ch07\fig07_07.cpp(8) : error C2166: l-value specifies
   const object
```

GNU C++ compiler error message:

```
fig07_07.cpp:6: error: uninitialized const `x'
Fig07_07.cpp:8: error: assignment of read-only variable `x'
```

图 7.7　常量变量必须被初始化

在需要常量表达式的地方都可以使用常量变量。在图 7.5 中，第 15 行的常量变量 arraySize 指定了数组大小。



常见的编程错误 7.6
只能使用常量声明自动和 static 数组的大小，如果不这样做，会导致编译错误。

使用常量变量指定数组大小可以使程序具有更好的伸缩性。在图 7.5 中，只需要简单地在 arraySize 的声明中把它的值从 10 改成 1000，就可以使第一条 for 语句填充一个有 1000 个元素的数组。如果没有使用常量变量 arraySize，那么为了将程序调整为可处理 1000 个数组元素，我们不得不修改程序的第 15、17 和 23 行。随着程序规模的扩大，这个方法更有利于编写清晰、易于修改的程序。



软件工程知识 7.1

把每个数组的大小定义为常量变量，而不是字面上的常量，可以使程序具有更好的伸缩性。



良好的编程习惯 7.2

把每个数组的大小定义为常量变量，而不是字面上的常量，可使程序更清晰。这个方法消除了所谓的魔法数字（magic numbers）。例如，在处理有 10 个元素的数组的代码中，多次提及数组大小 10，会给数字 10 一个人为的意义，当这个程序包含其他一些和数组大小无关的数字 10 时，就可能把读者搞糊涂。

求数组元素之和

数组的元素通常表示计算中用到的一系列值。例如，如果一个数组的元素表示考试成绩，那么教授可能希望累加数组元素，再利用总和计算考试的平均成绩。本章后面使用GradeBook类的例子，即图 7.16、图 7.17、图 7.23 和图 7.24，就使用了这个方法。

图 7.8 的程序计算了有 10 个元素的整数数组 a 所包含的值之和。程序在第 10 行声明、创建和初始化这个数组。for 语句（第 14~15 行）进行计算。数组 a 的初始化值可以通过用户在键盘键入、或从磁盘文件读入到程序。例如，for 语句：

```
for ( int j = 0 ; j < arraySize; j++ )
   cin >> a[ j ];
```

从键盘一次读取一个值并将之存储在元素 a[j]中。

```
1  // Fig. 7.8: fig07_08.cpp
2  // Compute the sum of the elements of the array.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     const int arraySize = 10; // constant variable indicating size of array
10    int a[ arraySize ] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11    int total = 0;
12
13    // sum contents of array a
14    for ( int i = 0; i < arraySize; i++ )
15       total += a[ i ];
16
17    cout << "Total of array elements: " << total << endl;
18
19    return 0; // indicates successful termination
20 } // end main
```

```
Total of array elements: 849
```

图 7.8　计算数组元素之和

使用条形统计图表显示数组数据

很多程序以图形方式向用户展现数据。例如，在条形统计图表中，数值以条形的方式显示。在这种图表中，按照一定比例，较长的条形表示较大的数值。以图形的形式表示数值数据的一个简单方法是条形统计图表，其中由星号（*）组成的一个个长条表示每个数值。

教授们常常喜欢查看考试成绩的分布情况。他们可能把每段成绩的人数用图表示，使成绩的分布直观化。假设有分数 87、68、94、100、83、78、85、91、76 和 87。注意到只有一个 100 分，有两个 90 多分，四个 80 多分，两个 70 多分，一个 60 多分，没有低于 60 分的。我们的下一个程序（见图 7.9）把这些成绩分布数据存放到一个有 11 个元素的数组中，每个元素对应一个成绩段。例如，n[0]表示分数为 0~9 的人数，n[7]表示分数为 70~79 的人数，n[10]表示分数为 100 的人数。本章后面 GradeBook 类的两个版本（见图 7.16、图 7.17、图 7.23 和图 7.24）都包含了计算一组成绩中这些分数频率的代码。眼下，我们通过观察分数集手工创建这个数组。

```
1  // Fig. 7.9: fig07_09.cpp
2  // Bar chart printing program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    const int arraySize = 11;
13    int n[ arraySize ] = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
14
15    cout << "Grade distribution:" << endl;
16
17    // for each element of array n, output a bar of the chart
18    for ( int i = 0; i < arraySize; i++ )
19    {
20       // output bar labels ("0-9:", ..., "90-99:", "100:" )
21       if ( i == 0 )
22          cout << "  0-9: ";
23       else if ( i == 10 )
24          cout << "  100: ";
25       else
26          cout << i * 10 << "-" << ( i * 10 ) + 9 << ": ";
27
28       // print bar of asterisks
29       for ( int stars = 0; stars < n[ i ]; stars++ )
30          cout << '*';
31
32       cout << endl; // start a new line of output
33    } // end outer for
34
35    return 0; // indicates successful termination
36 } // end main
```

```
Grade distribution:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

图 7.9　条形统计图表打印程序

这个程序从该数组读取数据，然后以条形统计图表的方式表示信息。程序显示了每个成绩段，后跟一个星号条指出在该段内成绩的个数。为了标记每个条形，第 21~26 行根据当前计数器变量 i

的值输出一个分数段（如"70~79:"）。嵌套的 for 语句（第 29~30 行）输出条形。注意第 29 行的循环继续条件（stars < n[i]）。每当程序进行到内层的 for 时，循环从 0 到 n[i]计数，从而用数组 n 中的一个值决定显示的星号个数。这个例子中没有学生得到低于 60 的分数，因此 n[0]~n[5]为 0。所以，程序在前面的六个成绩段之后不显示星号。



常见的编程错误 7.7

虽然可以在一个for语句以及其内部嵌套的另一个for语句中使用同样的控制变量，但是这会引起混乱和导致逻辑错误。

把数组元素当做计数器使用

有时候，程序使用计数器变量总结数据，譬如说调查的结果。在图 6.9 的掷骰子程序中，当程序掷 6 000 000 次时，我们使用不同的计数器跟踪一个骰子各个面的出现次数。这个程序的数组版本如图 7.10 所示。

```
1  // Fig. 7.10: fig07_10.cpp
2  // Roll a six-sided die 6,000,000 times.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 #include <cstdlib>
11 using std::rand;
12 using std::srand;
13
14 #include <ctime>
15 using std::time;
16
17 int main()
18 {
19    const int arraySize = 7; // ignore element zero
20    int frequency[ arraySize ] = { 0 };
21
22    srand( time( 0 ) ); // seed random number generator
23
24    // roll die 6,000,000 times; use die value as frequency index
25    for ( int roll = 1; roll <= 6000000; roll++ )
26       frequency[ 1 + rand() % 6 ]++;
27
28    cout << "Face" << setw( 13 ) << "Frequency" << endl;
29
30    // output each array element's value
31    for ( int face = 1; face < arraySize; face++ )
32       cout << setw( 4 ) << face << setw( 13 ) << frequency[ face ]
33          << endl;
34
35    return 0; // indicates successful termination
36 } // end main
```

```
Face    Frequency
   1      1000167
   2      1000149
   3      1000152
   4       998748
   5       999626
   6      1001158
```

图 7.10　使用数组代替 switch 的掷骰子程序

图 7.10 使用数组 frequency（第 20 行）计算骰子每个面出现的次数。该程序第 26 行的单条语句代替了图 6.9 中第 30~52 行的 switch 语句。第 26 行使用一个随机值来决定在循环的每次迭代中应该增加 frequency 的哪一个元素。第 26 行的计算产生一个 1 到 6 之间的随机下标，因此数组 frequency 应该足够大，可以存储 6 个计数器。但是，我们使用了一个有 7 个元素的数组，其中忽略 frequency[0]——使骰子值为 1 的面增加 frequency[1]而不是 frequency[0]，会更合乎逻辑。因此，每个面的值用来作为数组 frequency 的下标。同时，我们把图 6.9 中第 56~61 行替换为对数组 frequency 的循环，用来输出结果（第 31~33 行）。

使用数组汇总调查结果

我们的下一个例子（见图 7.11）利用数组汇总调查中搜集到的数据结果。考虑下列问题的描述：

询问 40 个学生，要他们按照等级 1 到 10 评价学生食堂食物的质量（1 表示很差，10 表示很好）。把这 40 个评价值存放到一个整数数组中，然后汇总此民意调查的结果。

```
1  // Fig. 7.11: fig07_11.cpp
2  // Student poll program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    // define array sizes
13    const int responseSize = 40; // size of array responses
14    const int frequencySize = 11; // size of array frequency
15
16    // place survey responses in array responses
17    const int responses[ responseSize ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
18       10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
19       5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
20
21    // initialize frequency counters to 0
22    int frequency[ frequencySize ] = { 0 };
23
24    // for each answer, select responses element and use that value
25    // as frequency subscript to determine element to increment
26    for ( int answer = 0; answer < responseSize; answer++ )
27       frequency[ responses[ answer ] ]++;
28
29    cout << "Rating" << setw( 17 ) << "Frequency" << endl;
30
31    // output each array element's value
32    for ( int rating = 1; rating < frequencySize; rating++ )
33       cout << setw( 6 ) << rating << setw( 17 ) << frequency[ rating ]
34          << endl;
35
36    return 0; // indicates successful termination
37 } // end main
```

```
Rating        Frequency
     1                2
     2                2
     3                2
     4                2
     5                5
     6               11
     7                5
     8                7
     9                1
    10                3
```

图 7.11　投票分析程序

```
cin >> string2;
```

从键盘读入一个字符串到string2，并在用户输入的字符串的后面追加空字符。请注意，上述语句只是提供了数组的名字，并没有关于数组大小的信息。确保读入字符串的数组能够存放用户通过键盘键入的任意字符串，则是程序员的责任。在默认情况下，cin从键盘读入字符直到遇到第一个空白字符为止——不管数组的大小是多少。因此，用cin和>>输入数据可以越过数组末端插入数据（若要了解如何防止在字符数组有效范围外插入数据的内容，请参见8.13节。）



常见的编程错误 7.9

不为cin>>提供一个足够大的字符数组存储从键盘键入的字符串，可以导致程序中数据的丢失和其他严重的运行时错误。

可以用cout和<<输出表示以空字符结束的字符串的字符数组。下面的语句：

```
cout << string2;
```

打印数组string2。请注意，和cin>>一样，cout<<也不在意字符数组有多大。字符串的字符连续输出，直到遇到终止的空字符为止。注意：cin和cout都假定将字符数组作为以空字符结束的字符串来处理；cin和cout不为其他数组类型提供类似的输入和输出处理功能。

图7.12中示范了用字符串文字初始化一个字符数组，读入一个字符串到一个字符数组，将字符数组作为一个字符串打印，以及访问一个字符串的单个字符。

```
1  // Fig. 7.12: fig07_12.cpp
2  // Treating character arrays as strings.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  int main()
9  {
10    char string1[ 20 ]; // reserves 20 characters
11    char string2[] = "string literal"; // reserves 15 characters
12
13    // read string from user into array string1
14    cout << "Enter the string \"hello there\": ";
15    cin >> string1; // reads "hello" [space terminates input]
16
17    // output strings
18    cout << "string1 is: " << string1 << "\nstring2 is: " << string2;
19
20    cout << "\nstring1 with spaces between characters is:\n";
21
22    // output characters until null character is reached
23    for ( int i = 0; string1[ i ] != '\0'; i++ )
24       cout << string1[ i ] << ' ';
25
26    cin >> string1; // reads "there"
27    cout << "\nstring1 is: " << string1 << endl;
28
29    return 0; // indicates successful termination
30 } // end main
```

```
Enter the string "hello there": hello there
string1 is: hello
string2 is: string literal
string1 with spaces between characters is:
h e l l o
string1 is: there
```

图7.12 作为字符串处理的字符数组

图7.12中的第23~24行采用一条for语句在数组string1中循环，并以空格间隔打印出各个字符。for 语句中的条件 string1[i]!='\0' 总是真的，直到循环遇到字符串的终止空字符为止。

static 局部数组和自动局部数组

第 6 章讨论了存储类别说明符 static。函数定义中的 static 局部变量在程序运行期间一直存在，但只在该函数体内可见。



性能提示 7.2

我们可以将 static 应用于局部数组声明，那样，数组就不会在每次程序调用该函数时都进行创建和初始化，也不会在程序中每次该函数结束时被销毁。这样可以提高性能，特别是在使用大型数组时。

当第一次遇到static局部数组的声明时，程序就初始化它们。如果一个static数组没有被程序员进行显式的初始化，那么在这个数组被创建时，编译器会把它的每个元素初始化为 0。回想一下，C++ 对自动变量是不执行这种默认初始化的。

图7.13示范了带有一个static局部数组（第28行）的函数staticArrayInit（第25~41行）和带有一个自动局部数组（第47行）的函数automaticArrayInit（第44~60行）。

```
1  // Fig. 7.13: fig07_13.cpp
2  // Static arrays are initialized to zero.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void staticArrayInit( void ); // function prototype
8  void automaticArrayInit( void ); // function prototype
9
10 int main()
11 {
12    cout << "First call to each function:\n";
13    staticArrayInit();
14    automaticArrayInit();
15
16    cout << "\n\nSecond call to each function:\n";
17    staticArrayInit();
18    automaticArrayInit();
19    cout << endl;
20
21    return 0; // indicates successful termination
22 } // end main
23
24 // function to demonstrate a static local array
25 void staticArrayInit( void )
26 {
27    // initializes elements to 0 first time function is called
28    static int array1[ 3 ]; // static local array
29
30    cout << "\nValues on entering staticArrayInit:\n";
31
32    // output contents of array1
33    for ( int i = 0; i < 3; i++ )
34       cout << "array1[" << i << "] = " << array1[ i ] << "  ";
35
36    cout << "\nValues on exiting staticArrayInit:\n";
37
38    // modify and output contents of array1
39    for ( int j = 0; j < 3; j++ )
```

图 7.13　static 数组的初始化和自动数组的初始化

```
40          cout << "array1[" << j << "] = " << ( array1[ j ] += 5 ) << "   ";
41    } // end function staticArrayInit
42
43    // function to demonstrate an automatic local array
44    void automaticArrayInit( void )
45    {
46       // initializes elements each time function is called
47       int array2[ 3 ] = { 1, 2, 3 }; // automatic local array
48
49       cout << "\n\nValues on entering automaticArrayInit:\n";
50
51       // output contents of array2
52       for ( int i = 0; i < 3; i++ )
53          cout << "array2[" << i << "] = " << array2[ i ] << "   ";
54
55       cout << "\nValues on exiting automaticArrayInit:\n";
56
57       // modify and output contents of array2
58       for ( int j = 0; j < 3; j++ )
59          cout << "array2[" << j << "] = " << ( array2[ j ] += 5 ) << "   ";
60    } // end function automaticArrayInit
```

```
First call to each function:

Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8
```

图 7.13　static 数组的初始化和自动数组的初始化（续）

函数 staticArrayInit 被调用了两次（第 13 和 17 行）。该函数第一次被调用时，编译器把 static 局部数组初始化为0。函数打印数组，给每个元素加5，并再次打印数组。该函数被第二次调用时，static 局部数组包含了在第一次函数调用期间所存储的修改值。函数 automaticArrayInit 也被调用了两次（第 14 和 18 行）。用值 1、2 和 3 初始化了这个自动局部数组的元素（第 47 行）。函数打印数组，给每个元素加 5，并再次打印数组。该函数第二次被调用时，数组元素被重新初始化为 1、2 和 3。这个数组具有自动存储类别，因此在每次调用 automaticArrayInit 时会被重新创建。



常见的编程错误 7.10

假定函数的局部 static 数组的元素在每次函数被调用时都被初始化，会导致程序的逻辑错误。

图7.14中的程序演示了在传递整个数组和传递一个数组元素之间的差别。第22~23行打印整数数组 a 的 5 个初始元素。第 28 行传递 a 和它的大小到函数 modifyArray（第 45~50 行）中，这个函数把 a 的每个元素都乘以 2（通过参数 b）。然后，第 32~33 行在 main 中重新打印数组 a。如输出所示，a 的元素确实被 modifyArray 修改了。接着，第 36 行打印标量 a[3]的值，然后第 38 行传递元素 a[3]到函数 modifyElement（第 54~58 行）中，这个函数把它的参数乘以 2 并打印新的值。请注意，因为单个的数组元素是按值传递的，所以当第 39 行在 main 中重新打印 a[3]时，这个值并没有被修改。

```
1  // Fig. 7.14: fig07_14.cpp
2  // Passing arrays and individual array elements to functions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 void modifyArray( int [], int ); // appears strange
11 void modifyElement( int );
12
13 int main()
14 {
15    const int arraySize = 5; // size of array a
16    int a[ arraySize ] = { 0, 1, 2, 3, 4 }; // initialize array a
17
18    cout << "Effects of passing entire array by reference:"
19       << "\n\nThe values of the original array are:\n";
20
21    // output original array elements
22    for ( int i = 0; i < arraySize; i++ )
23       cout << setw( 3 ) << a[ i ];
24
25    cout << endl;
26
27    // pass array a to modifyArray by reference
28    modifyArray( a, arraySize );
29    cout << "The values of the modified array are:\n";
30
31    // output modified array elements
32    for ( int j = 0; j < arraySize; j++ )
33       cout << setw( 3 ) << a[ j ];
34
35    cout << "\n\n\nEffects of passing array element by value:"
36       << "\n\na[3] before modifyElement: " << a[ 3 ] << endl;
37
38    modifyElement( a[ 3 ] ); // pass array element a[ 3 ] by value
39    cout << "a[3] after modifyElement: " << a[ 3 ] << endl;
40
41    return 0; // indicates successful termination
42 } // end main
43
44 // in function modifyArray, "b" points to the original array "a" in memory
45 void modifyArray( int b[], int sizeOfArray )
46 {
47    // multiply each array element by 2
48    for ( int k = 0; k < sizeOfArray; k++ )
49       b[ k ] *= 2;
50 } // end function modifyArray
51
```

图 7.14　传递数组和单个数组元素到函数中

```
52  // in function modifyElement, "e" is a local copy of
53  // array element a[ 3 ] passed from main
54  void modifyElement( int e )
55  {
56     // multiply parameter by 2
57     cout << "Value of element in modifyElement: " << ( e *= 2 ) << endl;
58  } // end function modifyElement
```

```
Effects of passing entire array by reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8


Effects of passing array element by value:

a[3] before modifyElement: 6
Value of element in modifyElement: 12
a[3] after modifyElement: 6
```

图 7.14　传递数组和单个数组元素到函数中（续）

在你的程序中，有时候不应该允许函数修改数组元素。C++ 提供类型限定词 const，用来防止在被调用函数的代码中修改调用者中的数组值。当一个函数的数组参数前用const限定词修饰时，这个数组的元素在函数体内就变成了常量，于是对函数体内任何修改此数组元素的企图都会导致编译错误。这使程序员能够防止在函数体内对数组元素无意识地进行修改。

图 7.15 示范了 const 限定词。函数 tryToModifyArray（第 21~26 行）用参数 const int b[]进行了定义，这表示数组 b 是常量，不能被修改。函数中有三处企图修改数组 b 的元素（第 23~25 行），都将导致编译错误。例如，Microsoft Visual C++.NET 的编译器会产生错误 "1-value specifies const object."。[注意：C++ 标准将 “对象” 定义为任何 “存储区域”，因此对象的概念包括了基本数据类型的变量和数组元素，也包括了类的实例（我们称为对象）。] 这个信息表示把一个 const 对象（例如 b[0]）作为一个左值使用是错误的——不能把一个 const 对象放到一个赋值运算符的左边来给它赋一个新的值。请注意，不同的编译器有不同的编译错误信息（如图 7.15 所示）。在第 10 章还将再次讨论 const 限定词。

```
1  // Fig. 7.15: fig07_15.cpp
2  // Demonstrating the const type qualifier.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void tryToModifyArray( const int [] ); // function prototype
8
9  int main()
10 {
11    int a[] = { 10, 20, 30 };
12
13    tryToModifyArray( a );
14    cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
15
16    return 0; // indicates successful termination
17 } // end main
18
```

图 7.15　给数组参数加上 const 类型限定词

```
// In function tryToModifyArray, "b" cannot be used
// to modify the original array "a" in main.
void tryToModifyArray( const int b[] )
{
   b[ 0 ] /= 2; // error
   b[ 1 ] /= 2; // error
   b[ 2 ] /= 2; // error
} // end function tryToModifyArray
```

Borland C++ 命令行编译器错误信息：

```
Error E2024 fig07_15.cpp 23: Cannot modify a const object
   in function tryToModifyArray(const int * const)
Error E2024 fig07_15.cpp 24: Cannot modify a const object
   in function tryToModifyArray(const int * const)
Error E2024 fig07_15.cpp 25: Cannot modify a const object
   in function tryToModifyArray(const int * const)
```

Microsoft Visual C++.NET 编译器错误信息：

```
C:\scpphtp5_examples\ch07\fig07_15.cpp(23) : error C2166: l-value specifies
   const object
C:\scpphtp5_examples\ch07\fig07_15.cpp(24) : error C2166: l-value specifies
   const object
C:\scpphtp5_examples\ch07\fig07_15.cpp(25) : error C2166: l-value specifies
   const object
```

GNU C++ 编译器错误信息：

```
fig07_15.cpp:23: error: assignment of read-only location
fig07_15.cpp:24: error: assignment of read-only location
fig07_15.cpp:25: error: assignment of read-only location
```

图 7.15　给数组参数加上 const 类型限定词（续）



常见的编程错误 7.11

调用者中的数组是以按引用方式传递的，因此可以在被调用函数中修改，忘记这一点可能会导致逻辑错误。



软件工程知识 7.3

在函数定义中给数组参数加上 const 类型限定词，以防止在函数体中修改原始的数组，这也是最小特权原则的一个例子。除非绝对需要，否则函数不应该有修改数组的权限。

7.6　实例研究：利用数组存放成绩的类 GradeBook

类 GradeBook 在第 3 章中引入，并在第 4 到第 6 章得到了扩展，这一节，我们进一步发展它。回想一下，这个类表示一个教授所用的成绩簿，用来存储和分析学生的成绩。该类前面的版本处理用户输入的成绩，但是不在类的数据成员中保存成绩。因此，重复的计算需要用户重新输入相同的成绩。解决这个问题的一个方法是把输入的每个成绩分别存储到该类的各个数据成员中。例如，我们可以在类 GradeBook 中创建数据成员 grade1、grade2、...、grade10 来存储 10 名学生的成绩。但是，对这些成绩进行求和并求其平均值的代码会比较麻烦。在这一节中，我们通过在一个数组中存储这些成绩来解决这个问题。

在类 GradeBook 的数组中存储学生成绩

这里介绍的 GradeBook 类版本（见图 7.16 和图 7.17）使用一个整数数组存储一次考试中若干学生的成绩。这样，就不需要重复输入相同的一组成绩。在图 7.16 的第 29 行，数组 grades 被声明为一个数据成员，因此，每个 GradeBook 对象包含它自己的成绩集。

```
1  // Fig. 7.16: GradeBook.h
2  // Definition of class GradeBook that uses an array to store test grades.
3  // Member functions are defined in GradeBook.cpp
4
5  #include <string> // program uses C++ Standard Library string class
6  using std::string;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    // constant -- number of students who took the test
13    const static int students = 10; // note public data
14
15    // constructor initializes course name and array of grades
16    GradeBook( string, const int [] );
17
18    void setCourseName( string ); // function to set the course name
19    string getCourseName(); // function to retrieve the course name
20    void displayMessage(); // display a welcome message
21    void processGrades(); // perform various operations on the grade data
22    int getMinimum(); // find the minimum grade for the test
23    int getMaximum(); // find the maximum grade for the test
24    double getAverage(); // determine the average grade for the test
25    void outputBarChart(); // output bar chart of grade distribution
26    void outputGrades(); // output the contents of the grades array
27 private:
28    string courseName; // course name for this grade book
29    int grades[ students ]; // array of student grades
30 }; // end class GradeBook
```

图 7.16　利用数组存储考试成绩的 GradeBook 类的定义

```
1  // Fig. 7.17: GradeBook.cpp
2  // Member-function definitions for class GradeBook that
3  // uses an array to store test grades.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8  using std::fixed;
9
10 #include <iomanip>
11 using std::setprecision;
12 using std::setw;
13
14 #include "GradeBook.h" // GradeBook class definition
15
16 // constructor initializes courseName and grades array
17 GradeBook::GradeBook( string name, const int gradesArray[] )
18 {
19    setCourseName( name ); // initialize courseName
20
21    // copy grades from gradeArray to grades data member
22    for ( int grade = 0; grade < students; grade++ )
```

图 7.17　操作成绩数组的 GradeBook 类的成员函数

```
23          grades[ grade ] = gradesArray[ grade ];
24    } // end GradeBook constructor
25
26    // function to set the course name
27    void GradeBook::setCourseName( string name )
28    {
29       courseName = name; // store the course name
30    } // end function setCourseName
31
32    // function to retrieve the course name
33    string GradeBook::getCourseName()
34    {
35       return courseName;
36    } // end function getCourseName
37
38    // display a welcome message to the GradeBook user
39    void GradeBook::displayMessage()
40    {
41       // this statement calls getCourseName to get the
42       // name of the course this GradeBook represents
43       cout << "Welcome to the grade book for\n" << getCourseName() << "!"
44          << endl;
45    } // end function displayMessage
46
47    // perform various operations on the data
48    void GradeBook::processGrades()
49    {
50       // output grades array
51       outputGrades();
52
53       // call function getAverage to calculate the average grade
54       cout << "\nClass average is " << setprecision( 2 ) << fixed <<
55          getAverage() << endl;
56
57       // call functions getMinimum and getMaximum
58       cout << "Lowest grade is " << getMinimum() << "\nHighest grade is "
59          << getMaximum() << endl;
60
61       // call function outputBarChart to print grade distribution chart
62       outputBarChart();
63    } // end function processGrades
64
65    // find minimum grade
66    int GradeBook::getMinimum()
67    {
68       int lowGrade = 100; // assume lowest grade is 100
69
70       // loop through grades array
71       for ( int grade = 0; grade < students; grade++ )
72       {
73          // if current grade lower than lowGrade, assign it to lowGrade
74          if ( grades[ grade ] < lowGrade )
75             lowGrade = grades[ grade ]; // new lowest grade
76       } // end for
77
78       return lowGrade; // return lowest grade
79    } // end function getMinimum
80
81    // find maximum grade
82    int GradeBook::getMaximum()
83    {
84       int highGrade = 0; // assume highest grade is 0
85
```

图 7.17　操作成绩数组的 GradeBook 类的成员函数（续）

```
86     // loop through grades array
87     for ( int grade = 0; grade < students; grade++ )
88     {
89        // if current grade higher than highGrade, assign it to highGrade
90        if ( grades[ grade ] > highGrade )
91           highGrade = grades[ grade ]; // new highest grade
92     } // end for
93
94     return highGrade; // return highest grade
95  } // end function getMaximum
96
97  // determine average grade for test
98  double GradeBook::getAverage()
99  {
100    int total = 0; // initialize total
101
102    // sum grades in array
103    for ( int grade = 0; grade < students; grade++ )
104       total += grades[ grade ];
105
106    // return average of grades
107    return static_cast< double >( total ) / students;
108 } // end function getAverage
109
110 // output bar chart displaying grade distribution
111 void GradeBook::outputBarChart()
112 {
113    cout << "\nGrade distribution:" << endl;
114
115    // stores frequency of grades in each range of 10 grades
116    const int frequencySize = 11;
117    int frequency[ frequencySize ] = { 0 };
118
119    // for each grade, increment the appropriate frequency
120    for ( int grade = 0; grade < students; grade++ )
121       frequency[ grades[ grade ] / 10 ]++;
122
123    // for each grade frequency, print bar in chart
124    for ( int count = 0; count < frequencySize; count++ )
125    {
126       // output bar labels ("0-9:", ..., "90-99:", "100:" )
127       if ( count == 0 )
128          cout << "  0-9: ";
129       else if ( count == 10 )
130          cout << "  100: ";
131       else
132          cout << count * 10 << "-" << ( count * 10 ) + 9 << ": ";
133
134       // print bar of asterisks
135       for ( int stars = 0; stars < frequency[ count ]; stars++ )
136          cout << '*';
137
138       cout << endl; // start a new line of output
139    } // end outer for
140 } // end function outputBarChart
141
142 // output the contents of the grades array
143 void GradeBook::outputGrades()
144 {
145    cout << "\nThe grades are:\n\n";
146
147    // output each student's grade
148    for ( int student = 0; student < students; student++ )
149       cout << "Student " << setw( 2 ) << student + 1 << ": " << setw( 3 )
150          << grades[ student ] << endl;
151 } // end function outputGrades
```

图 7.17　操作成绩数组的 GradeBook 类的成员函数（续）

```
1  // Fig. 7.18: fig07_18.cpp
2  // Creates GradeBook object using an array of grades.
3
4  #include "GradeBook.h" // GradeBook class definition
5
6  // function main begins program execution
7  int main()
8  {
9     // array of student grades
10    int gradesArray[ GradeBook::students ] =
11       { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
12
13    GradeBook myGradeBook(
14       "CS101 Introduction to C++ Programming", gradesArray );
15    myGradeBook.displayMessage();
16    myGradeBook.processGrades();
17    return 0;
18 } // end main
```

```
Welcome to the grade book for
CS101 Introduction to C++ Programming!

The grades are:

Student  1:  87
Student  2:  68
Student  3:  94
Student  4: 100
Student  5:  83
Student  6:  78
Student  7:  85
Student  8:  91
Student  9:  76
Student 10:  87

Class average is 84.90
Lowest grade is 68
Highest grade is 100

Grade distribution:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

图7.18　用一个成绩数组创建一个GradeBook对象，然后调用成员函数processGrades分析之

测试GradeBook类

图7.18的程序用整数数组gradesArray（在第10~11行声明和初始化）创建GradeBook类的一个对象（见图7.16和图7.17）。请注意，我们在表达式“GradeBook::students”（第10行）中使用二元作用域分辨运算符（::）来访问GradeBook类的static常量students。在此我们用这个常量创建一个数组，使其大小和类GradeBook中数据成员grades数组的大小相同。第13~14行将一个课程名称和gradesArray传递给GradeBook构造函数。第15行显示一条欢迎信息，第16行调用GradeBook对象的processGrades成员函数。输出显示了myGradeBook中10个成绩的汇总情况。

7.7 采用线性查找法查找数组

程序员通常要处理存储在数组中的大量数据。判断一个数组是否含有与某个关键值(key value) 相匹配的值，或许是必要的。在一个数组中发现一个特定元素的过程称为查找（searching）。这一节我们讨论简单的线性查找。本章后面的习题7.33要求你实现线性查找的一个递归版本。若要学习更复杂的、也更有效率的二分查找，请访问 en.wikipedia.org/wiki/Binary_search。

线性查找

线性查找法（见图 7.19 中的第 37~44 行）用一个查找关键值（search key）（第 40 行）与数组中的每个元素相比较。因为数组中的值是无序的，所以可能在第一个元素中找到这个值，也可能在最后一个元素中找到这个值。因此，平均起来，程序必须将查找关键值和数组的一半元素进行比较。若要判断该值不在数组中，程序必须把查找关键值和数组的每个元素进行比较。

```
1  // Fig. 7.19: fig07_19.cpp
2  // Linear search of an array.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  int linearSearch( const int [], int, int ); // prototype
9
10 int main()
11 {
12    const int arraySize = 100; // size of array a
13    int a[ arraySize ]; // create array a
14    int searchKey; // value to locate in array a
15
16    for ( int i = 0; i < arraySize; i++ )
17       a[ i ] = 2 * i; // create some data
18
19    cout << "Enter integer search key: ";
20    cin >> searchKey;
21
22    // attempt to locate searchKey in array a
23    int element = linearSearch( a, searchKey, arraySize );
24
25    // display results
26    if ( element != -1 )
27       cout << "Found value in element " << element << endl;
28    else
29       cout << "Value not found" << endl;
30
31    return 0; // indicates successful termination
32 } // end main
33
34 // compare key to every element of array until location is
35 // found or until end of array is reached; return subscript of
36 // element if key or -1 if key not found
37 int linearSearch( const int array[], int key, int sizeOfArray )
38 {
39    for ( int j = 0; j < sizeOfArray; j++ )
40       if ( array[ j ] == key ) // if found,
41          return j; // return location of key
```

图 7.19 数组的线性查找

```
42
43     return -1; // key not found
44  } // end function linearSearch
```

```
Enter integer search key: 36
Found value in element 18
```

```
Enter integer search key: 37
Value not found
```

图 7.19　数组的线性查找（续）

线性查找方法适用于小型数组或者未排序的数组（即元素不按特定次序排列的数组）。但是，对于大型的数组，线性查找是低效的。如果数组是有序的（例如，元素按升序排列），可以用高速的二分查找方法（en.wikipedia.org/wiki/Binary_search）进行查找。

7.8　采用插值排序法排序数组

排序数据（即按照某种特定的次序，譬如升序或降序排列数据）是最重要的计算应用之一。银行把所有支票按照账号排序，这样就可以在每个月底制定银行报表。电话公司把它们的电话目录按照姓名排序，以便于查找电话号码。事实上，每个组织都要对一些数据进行排序，而且在多数情况下，这些数据量是很大的。排序数据是一个有趣的问题，已经在计算机科学领域吸引了一批人进行了非常深入的研究。在这一章，我们讨论一个简单的排序策略。在习题中，我们探讨更复杂的、有更好性能的策略。



性能提示 7.4

简单的算法往往性能较差。它们的优点是易于编写、测试和调试。要实现最佳的性能，往往需要较复杂的算法。

插入排序

图 7.20 中的程序对一个具有 10 个元素的数组 data 进行升序排序。我们使用的方法称为插入排序—— 一种简单的、但是效率低的排序算法。此算法的第一次迭代取第二个元素，并且如果它小于第一个元素，那么把两者交换（即程序把第二个元素插入到第一个元素之前）。第二次迭代取第三个元素，和前两个元素比较后插入到恰当的位置，因此，这三个元素是有序的了。在算法的第 i 次迭代时，原始数组的前 i 个元素将排好序。

图 7.20 中的第 13 行声明并用下面的值初始化数组 data：

```
34   56   4   10   77   51   93   30   5   52
```

程序首先检查 data[0]和 data[1]，它们的值分别为 34 和 56。这两个元素已经有序了，所以程序继续执行——如果它们无序，程序就交换它们。

在第二次迭代时，程序检查 data[2]的值，该值是 4。这个值小于 56，所以程序把 4 存储在一个临时变量中，并把 56 往右移动一个元素。然后，程序检查并确定 4 小于 34，因此它把 34 往右移动一个元素。现在程序到达数组的开始处了，所以它把 4 放到 data[0]中。此时数组为

```
4   34   56   10   77   51   93   30   5   52
```

在第三次迭代时，程序把data[3]的值10存储到一个临时变量中。然后程序比较10和56，由于56比10大，因此把56往右移动一个元素。程序接着比较10和34，因为34也比10大，所以把34往右移动一个元素。当程序比较10和4时，发现10比4大，所以把10放到data[1]中。此时数组为

```
4  10  34  56  77  51  93  30  5  52
```

```
// Fig. 7.20: fig07_20.cpp
// This program sorts an array's values into ascending order.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

int main()
{
   const int arraySize = 10; // size of array a
   int data[ arraySize ] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
   int insert; // temporary variable to hold element to insert

   cout << "Unsorted array:\n";

   // output original array
   for ( int i = 0; i < arraySize; i++ )
      cout << setw( 4 ) << data[ i ];

   // insertion sort
   // loop over the elements of the array
   for ( int next = 1; next < arraySize; next++ )
   {
      insert = data[ next ]; // store the value in the current element

      int moveItem = next; // initialize location to place element

      // search for the location in which to put the current element
      while ( ( moveItem > 0 ) && ( data[ moveItem - 1 ] > insert ) )
      {
         // shift element one slot to the right
         data[ moveItem ] = data[ moveItem - 1 ];
         moveItem--;
      } // end while

      data[ moveItem ] = insert; // place inserted element into the array
   } // end for

   cout << "\nSorted array:\n";

   // output sorted array
   for ( int i = 0; i < arraySize; i++ )
      cout << setw( 4 ) << data[ i ];

   cout << endl;
   return 0; // indicates successful termination
} // end main
```

```
Unsorted array:
  34  56   4  10  77  51  93  30   5  52
Sorted array:
   4   5  10  30  34  51  52  56  77  93
```

图 7.20　用插入排序法排序数组

使用这个算法，在第*i*次迭代时，原始数组的前*i*个元素会被排序。但是，它们不一定是在最终位置上，因为在后面的迭代中，更小的值可能被定位。

排序由第24~39行的for语句执行，它循环遍历数组的元素。在每次迭代时，第26行把将要插入到数组的有序部分的元素值存储到临时变量insert（在第14行声明）中。第28行声明和初始化变量moveItem，它跟踪插入元素的位置。第31~36行循环定位元素应该插入的正确位置。当程序到达数组的开始处或者遇到比插入值小的元素时，循环终止。第34行往右移动一个元素，第35行把插入下一个元素的位置减1。当while循环终止后，第38行插入元素。当第24~39行的for语句终止时，数组的所有元素都被排序了。

插入排序的主要优点是易于编码，但是，它的速度慢。当对大型数组排序时，这一点变得更为明显。在习题中，我们将探讨其他一些数组排序算法。

7.9　多维数组

具有二维的多维数组通常用来表示数值表（tables of values），数值表包含的信息是按照行和列排列的。为了标识一个特定的表元素，我们必须指定两个下标。依据惯例，第一个下标表示元素的行，第二个下标表示元素的列。需要两个下标来标识特定元素的数组称为二维数组。注意，多维数组可以有多于二的维数（即下标）。图7.21展示了一个二维数组a。这个数组包含三行四列，因此它是一个3 × 4的数组。一般来讲，一个具有*m*行*n*列的数组称为*m* × *n*数组（*m*-by-*n* array）。

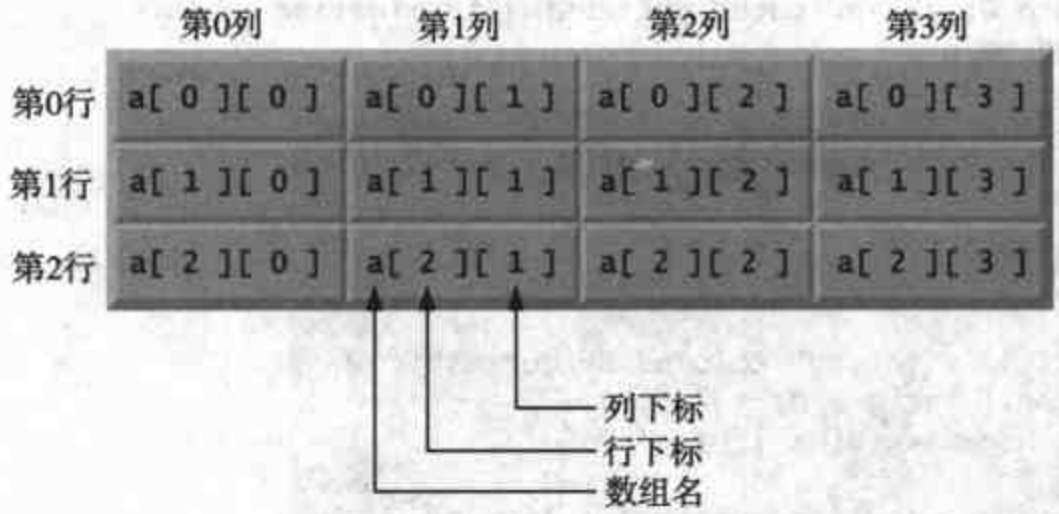


图7.21　具有三行四列的二维数组

在图7.21中，数组a中的每个元素都用a[i][j]形式的元素名来标识，其中a是数组名，i和j是下标，它们惟一地表示a中的每个元素。请注意，在第0行的所有元素名的第一个下标都是0；在第3列的所有元素名的第二个下标都是3。



常见的编程错误7.12

用a[x,y]不正确地引用一个二维数组元素a[x][y]是一个错误。事实上，由于C++计算表达式x,y（包含一个逗号运算符）得到y（逗号分隔表达式的最后一项），所以a[x,y]被认为是a[y]。

和一维数组类似，二维数组也可以在声明时被初始化。例如，一个二维数组b，它的第0行元素值为1和2，第1行元素值为3和4，这个数组可以被声明并初始化如下：

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

这些值用花括号按行分组。因此，1和2分别初始化b[0][0]和b[0][1]，3和4分别初始化b[1][0]和b[1][1]。如果一行没有足够的初始化值，那么该行剩余的元素都被初始化为0。因此，如下声明：

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

初始化b[0][0]为1、b[0][1]为0、b[1][0]为3、b[1][1]为4。

图7.22示范了在声明中初始化一个二维数组。第11~13行声明3个数组，每个数组有两行三列。

```
1  // Fig. 7.22: fig07_22.cpp
2  // Initializing multidimensional arrays.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void printArray( const int [][ 3 ] ); // prototype
8
9  int main()
10 {
11    int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
12    int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
13    int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
14
15    cout << "Values in array1 by row are:" << endl;
16    printArray( array1 );
17
18    cout << "\nValues in array2 by row are:" << endl;
19    printArray( array2 );
20
21    cout << "\nValues in array3 by row are:" << endl;
22    printArray( array3 );
23    return 0; // indicates successful termination
24 } // end main
25
26 // output array with two rows and three columns
27 void printArray( const int a[][ 3 ] )
28 {
29    // loop through array's rows
30    for ( int i = 0; i < 2; i++ )
31    {
32       // loop through columns of current row
33       for ( int j = 0; j < 3; j++ )
34          cout << a[ i ][ j ] << ' ';
35
36       cout << endl; // start new line of output
37    } // end outer for
38 } // end function printArray
```

```
Values in array1 by row are:
1 2 3
4 5 6

Values in array2 by row are:
1 2 3
4 5 0

Values in array3 by row are:
1 2 0
4 0 0
```

图7.22 多维数组的初始化

array1的声明（第11行）用两个子列表提供了6个初始化值。第一个子列表把数组的第0行初始化为1、2和3；第二个子列表把数组的第1行初始化为4、5和6。如果在array1的初始化列表中把两个子列表的花括号去掉，编译器先初始化第0行元素，再初始化第1行元素，会得到同样的结果。

```
1  // Fig. 7.23: GradeBook.h
2  // Definition of class GradeBook that uses a
3  // two-dimensional array to store test grades.
4  // Member functions are defined in GradeBook.cpp
5  #include <string> // program uses C++ Standard Library string class
6  using std::string;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    // constants
13    const static int students = 10; // number of students
14    const static int tests = 3; // number of tests
15
16    // constructor initializes course name and array of grades
17    GradeBook( string, const int [][ tests ] );
18
19    void setCourseName( string ); // function to set the course name
20    string getCourseName(); // function to retrieve the course name
21    void displayMessage(); // display a welcome message
22    void processGrades(); // perform various operations on the grade data
23    int getMinimum(); // find the minimum grade in the grade book
24    int getMaximum(); // find the maximum grade in the grade book
25    double getAverage( const int [], const int ); // find average of grades
26    void outputBarChart(); // output bar chart of grade distribution
27    void outputGrades(); // output the contents of the grades array
28 private:
29    string courseName; // course name for this grade book
30    int grades[ students ][ tests ]; // two-dimensional array of grades
31 }; // end class GradeBook
```

图 7.23　用二维数组存储成绩的类 GradeBook 的定义

```
1  // Fig. 7.24: GradeBook.cpp
2  // Member-function definitions for class GradeBook that
3  // uses a two-dimensional array to store grades.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8  using std::fixed;
9
10 #include <iomanip> // parameterized stream manipulators
11 using std::setprecision; // sets numeric output precision
12 using std::setw; // sets field width
13
14 // include definition of class GradeBook from GradeBook.h
15 #include "GradeBook.h"
16
17 // two-argument constructor initializes courseName and grades array
18 GradeBook::GradeBook( string name, const int gradesArray[][ tests ] )
19 {
20    setCourseName( name ); // initialize courseName
21
22    // copy grades from gradeArray to grades
23    for ( int student = 0; student < students; student++ )
24
25       for ( int test = 0; test < tests; test++ )
26          grades[ student ][ test ] = gradesArray[ student ][ test ];
27 } // end two-argument GradeBook constructor
28
```

图 7.24　操作二维成绩数组的 GradeBook 类成员函数的定义

```
29  // function to set the course name
30  void GradeBook::setCourseName( string name )
31  {
32     courseName = name; // store the course name
33  } // end function setCourseName
34
35  // function to retrieve the course name
36  string GradeBook::getCourseName()
37  {
38     return courseName;
39  } // end function getCourseName
40
41  // display a welcome message to the GradeBook user
42  void GradeBook::displayMessage()
43  {
44     // this statement calls getCourseName to get the
45     // name of the course this GradeBook represents
46     cout << "Welcome to the grade book for\n" << getCourseName() << "!"
47        << endl;
48  } // end function displayMessage
49
50  // perform various operations on the data
51  void GradeBook::processGrades()
52  {
53     // output grades array
54     outputGrades();
55
56     // call functions getMinimum and getMaximum
57     cout << "\nLowest grade in the grade book is " << getMinimum()
58        << "\nHighest grade in the grade book is " << getMaximum() << endl;
59
60     // output grade distribution chart of all grades on all tests
61     outputBarChart();
62  } // end function processGrades
63
64  // find minimum grade
65  int GradeBook::getMinimum()
66  {
67     int lowGrade = 100; // assume lowest grade is 100
68
69     // loop through rows of grades array
70     for ( int student = 0; student < students; student++ )
71     {
72        // loop through columns of current row
73        for ( int test = 0; test < tests; test++ )
74        {
75           // if current grade less than lowGrade, assign it to lowGrade
76           if ( grades[ student ][ test ] < lowGrade )
77              lowGrade = grades[ student ][ test ]; // new lowest grade
78        } // end inner for
79     } // end outer for
80
81     return lowGrade; // return lowest grade
82  } // end function getMinimum
83
84  // find maximum grade
85  int GradeBook::getMaximum()
86  {
87     int highGrade = 0; // assume highest grade is 0
88
89     // loop through rows of grades array
90     for ( int student = 0; student < students; student++ )
```

图 7.24　操作二维成绩数组的 GradeBook 类成员函数的定义（续）

```
91       {
92          // loop through columns of current row
93          for ( int test = 0; test < tests; test++ )
94          {
95             // if current grade greater than lowGrade, assign it to highGrade
96             if ( grades[ student ][ test ] > highGrade )
97                highGrade = grades[ student ][ test ]; // new highest grade
98          } // end inner for
99       } // end outer for
100
101      return highGrade; // return highest grade
102   } // end function getMaximum
103
104   // determine average grade for particular set of grades
105   double GradeBook::getAverage( const int setOfGrades[], const int grades )
106   {
107      int total = 0; // initialize total
108
109      // sum grades in array
110      for ( int grade = 0; grade < grades; grade++ )
111         total += setOfGrades[ grade ];
112
113      // return average of grades
114      return static_cast< double >( total ) / grades;
115   } // end function getAverage
116
117   // output bar chart displaying grade distribution
118   void GradeBook::outputBarChart()
119   {
120      cout << "\nOverall grade distribution:" << endl;
121
122      // stores frequency of grades in each range of 10 grades
123      const int frequencySize = 11;
124      int frequency[ frequencySize ] = { 0 };
125
126      // for each grade, increment the appropriate frequency
127      for ( int student = 0; student < students; student++ )
128
129         for ( int test = 0; test < tests; test++ )
130            ++frequency[ grades[ student ][ test ] / 10 ];
131
132      // for each grade frequency, print bar in chart
133      for ( int count = 0; count < frequencySize; count++ )
134      {
135         // output bar label ("0-9:", ..., "90-99:", "100:" )
136         if ( count == 0 )
137            cout << "  0-9: ";
138         else if ( count == 10 )
139            cout << "  100: ";
140         else
141            cout << count * 10 << "-" << ( count * 10 ) + 9 << ": ";
142
143         // print bar of asterisks
144         for ( int stars = 0; stars < frequency[ count ]; stars++ )
145            cout << '*';
146
147         cout << endl; // start a new line of output
148      } // end outer for
149   } // end function outputBarChart
150
151   // output the contents of the grades array
152   void GradeBook::outputGrades()
```

图 7.24　操作二维成绩数组的 GradeBook 类成员函数的定义（续）

```
153 {
154    cout << "\nThe grades are:\n\n";
155    cout << "            "; // align column heads
156
157    // create a column heading for each of the tests
158    for ( int test = 0; test < tests; test++ )
159       cout << "Test " << test + 1 << "  ";
160
161    cout << "Average" << endl; // student average column heading
162
163    // create rows/columns of text representing array grades
164    for ( int student = 0; student < students; student++ )
165    {
166       cout << "Student " << setw( 2 ) << student + 1;
167
168       // output student's grades
169       for ( int test = 0; test < tests; test++ )
170          cout << setw( 8 ) << grades[ student ][ test ];
171
172       // call member function getAverage to calculate student's average;
173       // pass row of grades and the value of tests as the arguments
174       double average = getAverage( grades[ student ], tests );
175       cout << setw( 9 ) << setprecision( 2 ) << fixed << average << endl;
176    } // end outer for
177 } // end function outputGrades
```

图 7.24　操作二维成绩数组的 GradeBook 类成员函数的定义（续）

5 个成员函数（在图 7.23 中的第 23~27 行声明）执行处理这些成绩的数组操作。这些成员函数都类似于早先的 GradeBook 类的一维数组版本（见图 7.16 和图 7.17）中的相应函数。成员函数 getMinimum（在图 7.24 中的第 65~82 行定义）确定这个学期学生的最低成绩。成员函数 getMaximum（在图 7.24 中的第 85~102 行定义）确定这个学期学生的最高成绩。成员函数 getAverage（图 7.24 中的第 105~115 行）确定某个学生这学期的平均成绩。成员函数 outputBarChart（图 7.24 中的第 118~149 行）输出这学期所有学生成绩分布的条形统计图表。成员函数 outputGrades（图 7.24 中的第 152~177 行）按表格形式输出二维数组和每个学生的学期平均成绩。

成员函数 getMinimum、getMaximum、outputBarChart 和 outputGrades 都用嵌套的 for 语句循环遍历数组 grades。例如，考虑成员函数 getMinimum 中的嵌套 for 语句（第 70~79 行）。外层的 for 语句通过把 student（即行下标）设置为 0 开始，这样，在内层 for 语句的循环体中可以把第 0 行的元素和变量 lowGrade 相比较。内层的 for 语句循环遍历 grades 的某一行，并把每个成绩和 lowGrade 相比较。如果等成绩小于 lowGrade，就把 lowGrade 设为该成绩。然后，外层的 for 语句把行下标增加为 1。第 1 行的元素和变量 lowGrade 比较。外层的 for 语句再把行下标增加到 2，并把第 2 行的元素和变量 lowGrade 相比较。这个过程重复下去，直到遍历完 grades 的所有行。当该嵌套语句执行完成之后，lowGrade 包含该二维数组中的最低成绩。成员函数 getMaximum 的实现和成员函数 getMinimum 类似。

图 7.24 中的成员函数 outputBarChart 和图 7.17 的几乎一样。但是，为了打印整个学期的全部成绩分布情况，这个成员函数使用一条嵌套的 for 语句（第 127~130 行），根据二维数组中的所有成绩创建一个一维数组 frequency。这两个 outputBarChart 成员函数其余部分的代码是一样的，都是显示条形图。

成员函数 outputGrades（第 152~177 行）除了输出每个学生的学期平均成绩外，也使用嵌套的 for 语句来输出数组 grades 的值。图 7.25 中的输出显示了结果，这与教师实际使用的成绩簿中的表格形式非常相似。第 158~159 行打印对应每次考试的列标题。我们使用一个计数器控制的 for 语句，以便可以用一个数字标识每次的考试。类似地，第 164~176 行中的 for 语句首先利用一个标记每个学生的计数器变量来输出行标签（第 166 行）。尽管数组下标从 0 开始，请注意，第 159 和 166 行分别输出 test+1 和 student+1，以生成从 1 开始的考试编号和学生编号（见图 7.25）。第 169~170 行的

内层 for 语句利用外层 for 语句的计数器变量 student 来循环遍历数组 grades 中的特定行，并输出每个学生的考试成绩。最后，第 174 行通过将 grades（例如，grades[student]）的当前行传递给成员函数 getAverage，取得每个学生的学期平均成绩。

```
1  // Fig. 7.25: fig07_25.cpp
2  // Creates GradeBook object using a two-dimensional array of grades.
3
4  #include "GradeBook.h" // GradeBook class definition
5
6  // function main begins program execution
7  int main()
8  {
9     // two-dimensional array of student grades
10    int gradesArray[ GradeBook::students ][ GradeBook::tests ] =
11       { { 87, 96, 70 },
12         { 68, 87, 90 },
13         { 94, 100, 90 },
14         { 100, 81, 82 },
15         { 83, 65, 85 },
16         { 78, 87, 65 },
17         { 85, 75, 83 },
18         { 91, 94, 100 },
19         { 76, 72, 84 },
20         { 87, 93, 73 } };
21
22    GradeBook myGradeBook(
23       "CS101 Introduction to C++ Programming", gradesArray );
24    myGradeBook.displayMessage();
25    myGradeBook.processGrades();
26    return 0; // indicates successful termination
27 } // end main
```

```
Welcome to the grade book for
CS101 Introduction to C++ Programming!

The grades are:

            Test 1  Test 2  Test 3  Average
Student  1      87      96      70    84.33
Student  2      68      87      90    81.67
Student  3      94     100      90    94.67
Student  4     100      81      82    87.67
Student  5      83      65      85    77.67
Student  6      78      87      65    76.67
Student  7      85      75      83    81.00
Student  8      91      94     100    95.00
Student  9      76      72      84    77.33
Student 10      87      93      73    84.33

Lowest grade in the grade book is 65
Highest grade in the grade book is 100

Overall grade distribution:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

图 7.25　用一个二维成绩数组创建 GradeBook 对象，然后调用成员函数 processGrades 分析成绩

```
// Fig. 7.26: fig07_26.cpp
// Demonstrating C++ Standard Library class template vector.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include <iomanip>
using std::setw;

#include <vector>
using std::vector;

void outputVector( const vector< int > & ); // display the vector
void inputVector( vector< int > & ); // input values into the vector

int main()
{
   vector< int > integers1( 7 ); // 7-element vector< int >
   vector< int > integers2( 10 ); // 10-element vector< int >

   // print integers1 size and contents
   cout << "Size of vector integers1 is " << integers1.size()
      << "\nvector after initialization:" << endl;
   outputVector( integers1 );

   // print integers2 size and contents
   cout << "\nSize of vector integers2 is " << integers2.size()
      << "\nvector after initialization:" << endl;
   outputVector( integers2 );

   // input and print integers1 and integers2
   cout << "\nEnter 17 integers:" << endl;
   inputVector( integers1 );
   inputVector( integers2 );

   cout << "\nAfter input, the vectors contain:\n"
      << "integers1:" << endl;
   outputVector( integers1 );
   cout << "integers2:" << endl;
   outputVector( integers2 );

   // use inequality (!=) operator with vector objects
   cout << "\nEvaluating: integers1 != integers2" << endl;

   if ( integers1 != integers2 )
      cout << "integers1 and integers2 are not equal" << endl;

   // create vector integers3 using integers1 as an
   // initializer; print size and contents
   vector< int > integers3( integers1 ); // copy constructor

   cout << "\nSize of vector integers3 is " << integers3.size()
      << "\nvector after initialization:" << endl;
   outputVector( integers3 );

   // use overloaded assignment (=) operator
   cout << "\nAssigning integers2 to integers1:" << endl;
   integers1 = integers2; // integers1 is larger than integers2

   cout << "integers1:" << endl;
   outputVector( integers1 );
```

图 7.26 C++标准库类模板 vector

```
63     cout << "integers2:" << endl;
64     outputVector( integers2 );
65
66     // use equality (==) operator with vector objects
67     cout << "\nEvaluating: integers1 == integers2" << endl;
68
69     if ( integers1 == integers2 )
70        cout << "integers1 and integers2 are equal" << endl;
71
72     // use square brackets to create rvalue
73     cout << "\nintegers1[5] is " << integers1[ 5 ];
74
75     // use square brackets to create lvalue
76     cout << "\n\nAssigning 1000 to integers1[5]" << endl;
77     integers1[ 5 ] = 1000;
78     cout << "integers1:" << endl;
79     outputVector( integers1 );
80
81     // attempt to use out-of-range subscript
82     cout << "\nAttempt to assign 1000 to integers1.at( 15 )" << endl;
83     integers1.at( 15 ) = 1000; // ERROR: out of range
84     return 0;
85  } // end main
86
87  // output vector contents
88  void outputVector( const vector< int > &array )
89  {
90     size_t i; // declare control variable
91
92     for ( i = 0; i < array.size(); i++ )
93     {
94        cout << setw( 12 ) << array[ i ];
95
96        if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
97           cout << endl;
98     } // end for
99
100    if ( i % 4 != 0 )
101       cout << endl;
102 } // end function outputVector
103
104 // input vector contents
105 void inputVector( vector< int > &array )
106 {
107    for ( size_t i = 0; i < array.size(); i++ )
108       cin >> array[ i ];
109 } // end function inputVector
```

```
Size of vector integers1 is 7
vector after initialization:
           0           0           0           0
           0           0           0

Size of vector integers2 is 10
vector after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
```

图 7.26　C++ 标准库类模板 vector（续）

```
After input, the vectors contain:
integers1:
         1          2          3          4
         5          6          7

integers2:
         8          9         10         11
        12         13         14         15
        16         17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of vector integers3 is 7
vector after initialization:
         1          2          3          4
         5          6          7

Assigning integers2 to integers1:
integers1:
         8          9         10         11
        12         13         14         15
        16         17

integers2:
         8          9         10         11
        12         13         14         15
        16         17

Evaluating: integers1 == integers2
integers1 and integers2 are equal
integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
         8          9         10         11
        12       1000         14         15
        16         17

Attempt to assign 1000 to integers1.at( 15 )
abnormal program termination
```

图 7.26　C++标准库类模板 vector（续）

第23行使用vector的成员函数size来获得integers1的大小（即元素的个数）。第25行将integers1传递给函数outputVector（第88~102行），该函数使用方括号（[]）获取vector的每个元素的值，这个值是可以输出的。注意这种表示法和用来访问一个数组元素值的表示法的类似之处。第28和30行对integers2进行同样的操作。

类模板vector的成员函数size返回一个vector的元素个数，它是一个size_t类型的值（在许多系统中size_t代表unsigned int类型）。因此，第90行也把控制变量i声明为size_t类型。在一些编译器上，把i声明为int会使编译器发出一条警告消息，因为在循环继续条件（第92行）中会把一个有符号值（即int i）和一个无符号值（即函数size返回的size_t类型的值）进行比较。

第34~35行把integers1和integers2传递到函数inputVector（第105~109行），从用户处读入每个vector的元素值。函数inputVector使用方括号（[]）获取左值，该左值可用来存储vector每个元素的输入值。

第46行说明vector对象可以用!=运算符直接进行比较。如果两个vector的内容不相等，运算符返回true；否则，运算符返回false。

C++标准库类模板vector允许程序员创建一个新的vector对象，并用一个已有的vector的内容来初始化它。第51行创建一个vector对象（integers3），并用integers1的一个副本初始化它。这会调用vector所谓的拷贝构造函数来执行复制操作。第11章将详细地研究拷贝构造函数。第53和55行输出integers3的大小和内容，以说明它被正确地初始化了。

低的值。]图7.27显示这两个骰子值36种可能的组合。程序应该掷这两个骰子36 000次。请利用一个一维数组记录每个可能的和出现的次数。以表格的形式打印结果，并判定两数之和是否合理（例如，有6种方式可以掷到7，因此，所有掷出的和值，大约有1/6是7）。

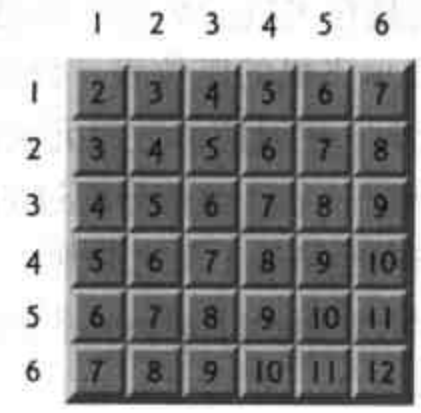


图 7.27　掷两个骰子的 36 种可能结果

7.18 请问下面的程序做了什么？

```
// Ex. 7.18: Ex07_18.cpp
// What does this program do?
#include <iostream>
using std::cout;
using std::endl;

int whatIsThis( int [], int ); // function prototype

int main()
{
   const int arraySize = 10;
   int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   int result = whatIsThis( a, arraySize );

   cout << "Result is " << result << endl;
   return 0; // indicates successful termination
} // end main

// What does this function do?
int whatIsThis( int b[], int size )
{
   if ( size == 1 ) // base case
      return b[ 0 ];
   else // recursive step
      return b[ size - 1 ] + whatIsThis( b, size - 1 );
} // end function whatIsThis
```

7.19 修改图6.11中的程序，玩1000次掷双骰子游戏。程序应能够跟踪统计结果，并回答以下问题：

a) 第1次、第2次、……、第20次以及第20次以后的游戏中，共赢了几次？

b) 第1次、第2次、……、第20次以及第20次以后的游戏中，共输了几次？

c) 掷双骰子游戏中赢的机率有多大？［注意：你会发现掷双骰子游戏是最公平的赌博游戏之一。请解释为什么？］

d) 掷双骰子游戏的平均时间为多长？

e) 玩的时间越长，是否意味着赢的机会越多？

7.20（航空订票系统）一家小型航空公司刚为它新的自动订票系统购买了一台计算机。要求你来编写新系统的程序。要写一个程序，为该航空公司惟一一架飞机的每次航班分配座位（此机的载容量：10个座位）。

程序应该显示如下的二选一菜单——Please type 1 for "First Class" 和 Please type 2 for "Economy"。如果乘客输入1，程序应该为其分配在头等舱的座位（座位1~5）。如果乘客输入2，程序应该为其分配在经济舱的座位（座位6~10）。程序应该打印一张登机牌，表明该用户的座位号以及它是在飞机的头等舱还是在经济舱。

使用一个一维数组表示飞机的座位表。数组的所有元素初始化为0，表明所有的座位是空的。当座位被分配时，把对应的数组元素设置为1，表明这个座位不再是空的了。

当然，程序不应该分配一个已经被分配过的座位。当头等舱满员时，程序应该询问用户，是否可以把他安排在经济舱（反之亦然）。如果可以，就进行恰当的座位分配。如果不可以，就打印消息 "Next flight leaves in 3 hours."。

7.21 请问下面的程序做了什么？

```
1  // Ex. 7.21: Ex07_21.cpp
2  // What does this program do?
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void someFunction( int [], int, int ); // function prototype
8
9  int main()
10 {
11    const int arraySize = 10;
12    int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13
14    cout << "The values in the array are:" << endl;
15    someFunction( a, 0, arraySize );
16    cout << endl;
17    return 0; // indicates successful termination
18 } // end main
19
20 // What does this function do?
21 void someFunction( int b[], int current, int size )
22 {
23    if ( current < size )
24    {
25       someFunction( b, current + 1, size );
26       cout << b[ current ] << "  ";
27    } // end if
28 } // end function someFunction
```

7.22 利用一个二维数组解决下面的问题。一家公司有4名售货员（1~4），卖5种不同的产品（1~5）。每天，对于每种不同产品的销售情况，每位售货员都要递交相应的一张纸条。每张纸条包含以下内容：

a) 售货员编号

b) 产品编号

c) 该产品当天的销售总额

因此，每位售货员每天会上交0到5张销售纸条。假设现在有上个月所有纸条的信息。请编写一个程序，读入上个月的销售信息，统计每位售货员每种产品的销售总额。所有的总额应存储在一个二维数组 sales 中。处理完上个月的信息后，以表格形式打印出结果，每列表示一位售货员，每行表示一种产品。统计每行求出上个月每种产品的销售总额；统计每列求出上个月每位售货员的销售总额。打印输出的表格应该在相应行的右边和相应列的下面显示这些统计结果。

动一格。而2类移动则是水平方向左移一格，垂直方向上移动两格。水平向左的移动和竖直向上的移动都用负数来表示。这8种可能的移动方式可以用两个一维数组horizontal和vertical表示，如下所示：

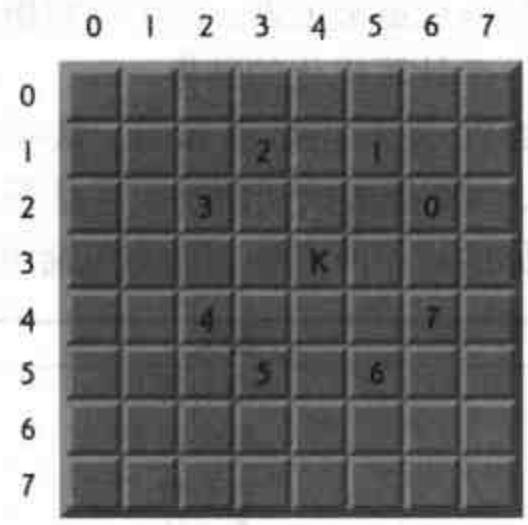


图7.29 骑士的8种可能的移动情况

```
horizontal[ 0 ] = 2
horizontal[ 1 ] = 1
horizontal[ 2 ] = -1
horizontal[ 3 ] = -2
horizontal[ 4 ] = -2
horizontal[ 5 ] = -1
horizontal[ 6 ] = 1
horizontal[ 7 ] = 2

vertical[ 0 ] = -1
vertical[ 1 ] = -2
vertical[ 2 ] = -2
vertical[ 3 ] = -1
vertical[ 4 ] = 1
vertical[ 5 ] = 2
vertical[ 6 ] = 2
vertical[ 7 ] = 1
```

令变量currentRow和currentColumn表示骑士当前位置的行和列。为了执行一次moveNumber类型的移动（其中moveNumber在0和7之间），程序应该使用如下语句：

```
currentRow += vertical[ moveNumber ];
currentColumn += horizontal[ moveNumber ];
```

定义一个从1到64变化的计数器。记录骑士在每一方格中移动的最近计数。记住，要检测每种可能的移动，以确定骑士是否已经访问过该方格，当然，也要检测每种可能的移动以保证骑士不会跑到棋盘外面。现在编写一个程序，在棋盘上移动骑士。运行这个程序，看看骑士移动了几次？

c) 在尝试编写和运行了一个骑士周游程序之后，你或许已经发现一些有用的想法了。我们将利用这些智慧，为骑士的移动开发一种试探法（或策略）。试探法不一定保证成功，但是一个精心研制的试探法会极大地增加成功的机会。你可能已经注意到外部的方格比靠近棋盘中心的方格更麻烦。事实上，最麻烦的或难以接近的方格就是四个角落。

最直观的想法是应该先把骑士移动到最难到达的方格，将最容易到达的空出来，这样，在接近周游末期棋盘变得拥挤时，成功的机会就更大。

我们可以开发一个“可达性试探法”，根据每个方格的可到达程度将它们分类，然后总是把骑士移动到最难到达的那个方格（当然，要符合骑士的L形移动规则）。我们

空棋盘上有多少方格可以被“排除”。棋盘四个角落的值都是22，如图7.30所示。]一旦在64个方格中都放入“排除数”后，一个适当的试探法可以是：把下一个皇后放在具有最小排除数的方格中。为什么会凭直觉选用这个策略？



图7.30 放一个皇后在左上角后，要排除的22个方格

7.27（八皇后问题：蛮力方法）在这道习题中，你将开发几个蛮力方法来解决习题7.26所介绍的八皇后问题。

a) 利用习题7.25中介绍的随机蛮力方法，解决八皇后问题。

b) 使用穷举法，即尝试八个皇后在棋盘上的所有可能组合。

c) 为什么穷举法不适宜解决骑士周游问题？

d) 全面比较随机蛮力方法和穷举蛮力方法。

7.28（骑士周游问题：封闭周游测试）在骑士周游问题中，当骑士移动64次并经过每个方格一次且只有一次时，才算是一个完整的周游。封闭周游是指骑士的第65次移动回到了出发点的周游。修改你在习题7.24中编写的骑士周游程序，当产生一个完整周游时，测试它是否是一个封闭周游。

7.29（Eratosthenes筛选法）质数是只能被1及其本身整除的数。Eratosthenes筛选法是一种寻找质数的方法。它的操作如下：

a) 创建一个数组，将它的所有元素都初始化为1（真）。下标为质数的数组元素将保持是1，其他元素最后都会被设置为0。在这道习题中，可以不考虑元素0和元素1。

b) 从数组下标2开始，每次找到一个值为1的数组元素时，对数组剩余部分循环，并将下标为该元素下标倍数的元素设置为0。对于数组下标2，数组中2之后的下标为2倍数的元素（下标4、6、8、10等）都被设置为0；对于数组下标3，数组中3之后的下标为3倍数的元素（下标6、9、12、15等）都被设置为0；依次类推。

当这个过程完成之后，仍然为1的数组元素，表明其下标就是一个质数。然后可以把这些下标打印出来。编写一个程序，利用一个1000个元素的数组，判断并打印2到999之间的质数。忽略数组的元素0。

7.30（桶排序）桶排序（bucket sort）从一个一维的待排序的正整数数组和一个二维整数数组开始，其中二维数组的行下标是从0到9，列下标是从0到n–1，n是一维数组中待排序值的个数。这个二维数组的每一行都称为一个桶。编写一个函数bucketSort，它采用一个整数数组和该数组的大小作为参数，并执行以下操作：

a) 对于一维数组的每个值，根据值的个位数，将其放到桶数组的各行中。例如，97放在第7行，3放在第3行，100放在第0行。这称为“分布过程”。

b) 在桶数组中逐行循环编历，并把值复制回原始数组。这称为“收集过程”。上述值在一维数组中的新次序是100、3和97。

8.1　简介

本章讨论C++程序设计语言的最强大功能之一，指针。在第6章，我们看到引用可用于实现按引用传递。指针也能够按引用传递，并且可以创建和操作动态的数据结构（即可以增长和缩减的数据结构），例如链表、队列、堆栈和树。本章解释基本的指针概念，重点强调数组和指针之间的密切关系。把数组看做指针的观点来自C程序设计语言。正如我们在第7章中看到的，C++标准库类vector以成熟对象的形式提供了数组的一种实现。

类似地，C++实际上提供了两种类型的字符串——string类对象（我们从第3章开始就已在使用）和C风格的、基于指针的char*字符串。本章讨论char*字符串以加深你对指针的理解。事实上，我们在7.4节介绍并在图7.12中使用的以空字符结尾的字符串都是基于指针的char*字符串。本章也包含了丰富的使用char*字符串的基本字符串处理练习。在遗留的C和C++系统中，广泛使用的是C风格的基于指针的char*字符串。因此，如果使用的是遗留的C或C++系统，就需要操作这些基于指针的char*字符串。

第13章我们将通过类继续审视指针的作用，将看到所谓面向对象编程的“多态处理”是用指针和引用实现的。

8.2　指针变量的声明和初始化

指针变量把内存地址作为它们的值。通常，变量直接包含特定的值。但是，指针包含的是包含特定值的变量的内存地址，从这个意义上讲，变量名直接引用数值，而指针间接引用数值，如图8.1所示。通过指针引用数值称为间接引用。请注意，在图中，通常把指针表示为一个变量指向另一个变量的箭头，前者包含一个地址值，后者则是在内存中位于该地址的变量。

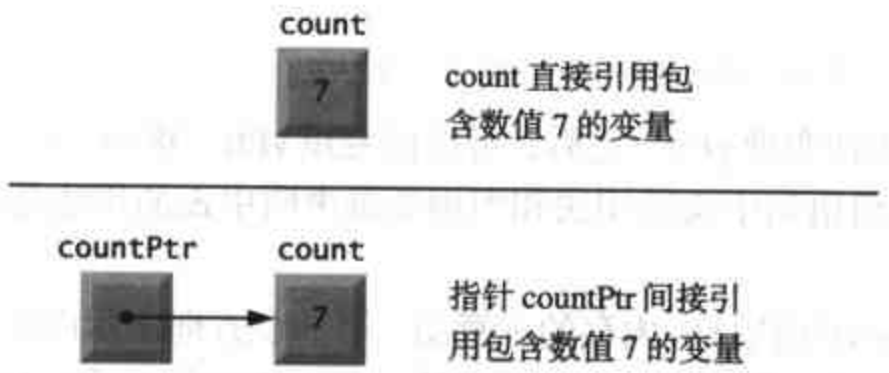


图 8.1　直接和间接引用变量

和其他变量一样，指针在使用前必须被声明。例如，对于图8.1中的指针，下面的声明：

```
int *countPtr, count;
```

声明了变量countPtr是int*类型的（即一个指向int值的指针），读做“countPtr是一个指向int的指针”或“countPtr指向一个int类型对象”。同时，该声明指出变量count是一个int类型的变量，而不是指向int的指针。声明中的*只应用于countPtr。每一个声明为指针的变量在其前面必须有一个星号（*）。例如，如下声明：

```
double *xPtr, *yPtr;
```

表示xPtr和yPtr都是指向double值的指针。声明中出现的*不是一个运算符；它只是表明被声明的变量是一个指针。指针可以被声明来指向任何数据类型的对象。



常见的编程错误 8.1

将用于声明一个指针的*误认为对声明中逗号分隔变量列表中的所有变量名都有效，这是错误的。声明时，每一个指针变量名前都必须有前缀*（之间空格可有可无——编译器忽略空格）。每个声明只声明一个变量，将有助于避免这种错误，同时提高程序的可读性。



良好的编程习惯 8.1

虽然不是必要的，但是，在指针变量名中包含字母“Ptr”，可以清楚地表明这些变量是指针，应该做适当的处理。

指针可以在声明时或在赋值语句中初始化。指针可以被初始化为0、NULL或地址。值为0或NULL的指针不指向任何对象，称为空指针（null pointer）。符号常量NULL在头文件<iostream>（和其他一些标准库头文件）中定义，表示数值0。把一个指针初始化为NULL和0是等价的，但是，在C++中约定使用0。0在赋值后被转换为一个适当类型的指针。只有整数0可以直接赋给一个指针变量，而其他整数要先强制转换成一个指针类型。在8.3节中讨论把一个变量的地址赋给一个指针。



错误预防技巧 8.1

初始化指针，以防止指向一个未知的或未被初始化的内存空间。

8.3 指针运算符

地址运算符&是一个一元运算符，返回操作数的内存地址。例如，假设有如下声明：

```
int y = 5; // declare variable y
int *yPtr; // declare pointer variable yPtr
```

则语句：

```
yPtr = &y; // assign address of y to yPtr
```

把变量y的地址赋值给指针变量yPtr。之后，可以说变量yPtr“指向”y。现在，yPtr间接引用变量y的值。请注意，上述赋值语句中&的用法和引用变量声明中&的用法是不一样的，后者的前面总是有一个数据类型名。

图8.2是前面的赋值语句执行后内存的示意图。第一个方框表示内存中的指针yPtr，第二个方框表示内存中的变量y，画一个从第一个方框到第二个方框的箭头表示这种“指向关系”。

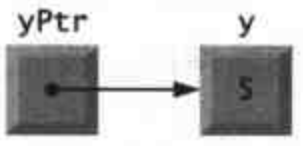


图8.2 内存中指针指向变量的示意图

图8.3是一个内存中指针的另外一种示意图，其中假设整数变量y存储在内存地址600000，指针变量yPtr存储在内存地址500000。地址运算符的操作数必须是左值（即可以被赋值的，如变量名或引用）；地址运算符不能用于常量或者结果不是引用的表达式。



图8.3 内存中y和yPtr的表示

*运算符通常称为间接运算符或间接引用运算符，它返回其指针操作数所指向的对象的同义词（即别名或者昵称）。例如（再次参见图 8.2），如下语句：

```
cout << *yPtr << endl;
```

打印变量 y 的值，即 5，和下面的语句：

```
cout << y << endl;
```

是一样的。以这种方式使用 *，称为间接引用一个指针。请注意，间接引用的指针也可以在赋值语句的左边使用，如

```
*yPtr = 9 ;
```

它把 9 赋给图 8.3 中的 y。间接引用的指针还可以用来接收输入的值，如

```
cin >> *yPtr;
```

它把输入的值存放到 y 中。间接引用的指针是左值。



常见的编程错误 8.2

间接引用一个未被正确初始化的、或未被赋值指向内存中具体位置的指针，可能会导致致命的运行期错误，或者意外修改了重要数据，以致程序运行结束后可能得到错误结果。



常见的编程错误 8.3

试图间接引用非指针变量，会导致编译错误。



常见的编程错误 8.4

间接引用空指针，通常是致命的运行期错误。

图 8.4 中的程序示范了 & 和 * 指针运算符的用法。在这个例子中，以十六进制整数的格式输出内存地址。（有关十六进制整数的更多内容，请参见附录 D。）请注意，该程序输出的十六进制内存地址决定于编译器和操作系统，所以，当运行程序时可能会得到不同的结果。

```
1  // Fig. 8.4: fig08_04.cpp
2  // Using the & and * operators.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int a; // a is an integer
10    int *aPtr; // aPtr is an int * -- pointer to an integer
11
12    a = 7; // assigned 7 to a
13    aPtr = &a; // assign the address of a to aPtr
14
15    cout << "The address of a is " << &a
16       << "\nThe value of aPtr is " << aPtr;
17    cout << "\n\nThe value of a is " << a
18       << "\nThe value of *aPtr is " << *aPtr;
19    cout << "\n\nShowing that * and & are inverses of "
20       << "each other.\n&*aPtr = " << &*aPtr
21       << "\n*&aPtr = " << *&aPtr << endl;
22    return 0; // indicates successful termination
23 } // end main
```

图 8.4　指针运算符 & 和 *

```
The address of a is 0012F580
The value of aPtr is 0012F580

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012F580
*&aPtr = 0012F580
```

图 8.4　指针运算符 & 和 *（续）



可移植性提示 8.1

指针的输出格式是与编译器相关的。一些系统以十六进制整数格式输出指针变量，而另一些以十进制整数格式输出。

请注意，a（第 15 行）的地址和 aPtr（第 16 行）的值在输出中是相同的，证实了 a 的地址确实赋给了指针变量 aPtr。& 和 * 运算符互为反运算——以任一顺序把它们两个连续应用到 aPtr，它们“彼此抵消”，打印出相同的结果（aPtr 中的值）。

图 8.5 列出了迄今为止所介绍的运算符的优先级和结合律。注意，地址运算符（&）和间接引用运算符（*）是表中第三层优先级中的一元运算符。

运算符	结合律	类型
() []	从左到右	最高
++ -- static_cast<*type*>(*operand*)	从左到右	一元（后置）
++ -- + - ! & *	从右到左	一元（前置）
* / %	从左到右	乘法
+ -	从左到右	加法
<< >>	从左到右	插入 / 提取
< <= > >=	从左到右	关系
== !=	从左到右	相等
&&	从左到右	逻辑“与”
\|\|	从左到右	逻辑“或”
?:	从右到左	条件
= += -= *= /= %=	从右到左	赋值
,	从左到右	逗号

图 8.5　运算符的优先级和结合律

8.4　用指针通过按引用传递方式将参数传递给函数

C++中有三种传递参数到函数的方法——按值传递、用引用参数按引用传递和用指针参数按引用传递。第6章比较了按值传递和用引用参数按引用传递。在这一节，我们解释用指针参数按引用传递。

我们在第 6 章中知道，可以用 return 从被调用函数返回一个值给调用者（或者从被调用函数返回控制权而不需回传一个值）。我们也知道可以用引用参数把参数传递到函数中。这种参数允许被调用函数修改参数在调用者中的原始值。引用参数也可以使程序向函数传递大型的数据对象，而避免按值传递对象所需要的开销（即需要复制对象的开销）。和引用一样，指针也可用于修改调用者中的一个或多个变量，或者可以将指向大型数据对象的指针传递给函数从而避免按值传递对象所需要的开销。

在 C++ 中，程序员可以用指针和间接运算符（*）来完成按引用传递（就像 C 程序中的按引用传递一样，因为C是没有引用的）。当调用一个函数，需要修改其实参时，就传递该参数的地址。通常这是通过在值需要修改的变量的名字前应用地址运算符（&）来实现的。

正如第 7 章所述，数组名就是数组在内存中的起始地址（即数组名已经是一个指针），所以不用运算符 & 传递数组。数组的名字 arrayName 等价于 &arrayName[0]。当一个变量的地址传递给函数时，可以在函数中用间接运算符（*）来构成该变量名的一个别名——它可以在调用者的内存位置修改该变量的值。

图 8.6 和图 8.7 介绍了计算整数的立方函数的两个版本——cubeByValue 和 cubeByReference。图 8.6 通过按值传递将变量 number 传递给函数 cubeByValue（第 15 行）。函数 cubeByValue（第 21~24 行）计算它的参数的立方，并用一个 return 语句把新值回传给 main（第 23 行）。该新值赋给了 main 中的 number（第 15 行）。请注意，在修改变量 number 的值之前，调用者可以检查函数调用的结果。例如，在这个程序中，我们可以把 cubeByValue 的结果存储到另一个变量，检查它的值，当判定返回值是合理的之后，再把结果赋给 number。

```
1  // Fig. 8.6: fig08_06.cpp
2  // Cube a variable using pass-by-value.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int cubeByValue( int ); // prototype
8
9  int main()
10 {
11    int number = 5;
12
13    cout << "The original value of number is " << number;
14
15    number = cubeByValue( number ); // pass number by value to cubeByValue
16    cout << "\nThe new value of number is " << number << endl;
17    return 0; // indicates successful termination
18 } // end main
19
20 // calculate and return cube of integer argument
21 int cubeByValue( int n )
22 {
23    return n * n * n; // cube local variable n and return result
24 } // end function cubeByValue
```

```
The original value of number is 5
The new value of number is 125
```

图 8.6　使用按值传递计算一个变量值的立方

图 8.7 用一个指针实参（第 15 行）以按引用传递把变量 number 传递给函数 cubeByReference——number 的地址被传递到函数中。函数 cubeByReference（第 22~25 行）指定参数 nPtr（一个指向 int 的指针）来接收它的实参。此函数间接引用这个指针，并计算 nPtr 所指的值的立方（第 24 行）。这直接改变了 main 中 number 的值。



常见的编程错误 8.5

当需要间接引用一个指针以获取该指针指向的值时，却不间接引用，这是错误的。

接收地址实参的函数，必须定义指针参数来接收地址。例如，函数 cubeByReference 的头部（第 22 行）指定 cubeByReference 接收一个 int 变量的地址（即指向一个 int 的一个指针）作为参数，把该地址存储到局部变量 nPtr，没有返回值。

cubeByReference 的函数原型（第 7 行）在圆括号中包含 int*。和其他变量类型一样，在函数原型中不需要包含指针参数的名字。基于文档目的，可以添加参数名，但它会被编译器忽略。

```
1  // Fig. 8.7: fig08_07.cpp
2  // Cube a variable using pass-by-reference with a pointer argument.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void cubeByReference( int * ); // prototype
8
9  int main()
10 {
11    int number = 5;
12
13    cout << "The original value of number is " << number;
14
15    cubeByReference( &number ); // pass number address to cubeByReference
16
17    cout << "\nThe new value of number is " << number << endl;
18    return 0; // indicates successful termination
19 } // end main
20
21 // calculate cube of *nPtr; modifies variable number in main
22 void cubeByReference( int *nPtr )
23 {
24    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
25 } // end function cubeByReference
```

```
The original value of number is 5
The new value of number is 125
```

图 8.7　用指针参数按引用传递计算一个变量值的立方

图 8.8 和图 8.9 用图表分别分析了图 8.6 和图 8.7 中程序的执行情况。



软件工程知识 8.1

使用按值传递把参数传递给函数，除非调用者明确要求被调用函数直接修改调用者中参数变量的值。这也是最小特权原则的一个例子。

第 1 步：在 main 调用 cubeByValue 之前：

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: 未定义

第 2 步：在 cubeByValue 收到调用之后：

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

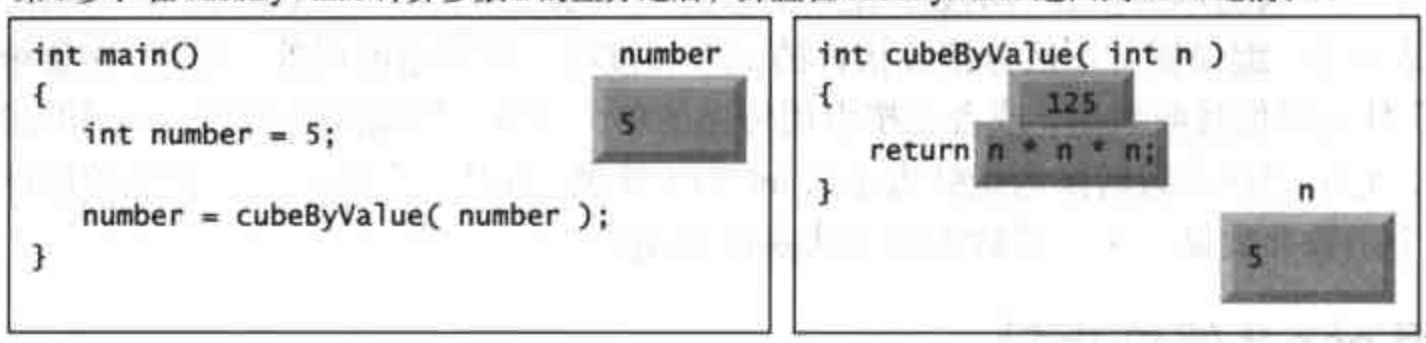
number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

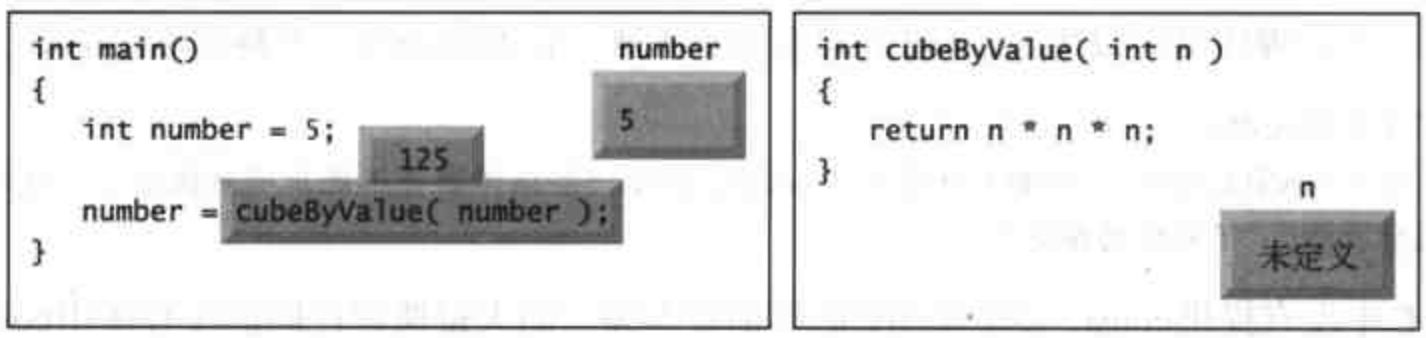
n: 5

图 8.8　对图 8.6 中程序按值传递的分析

第 3 步：在 cubeByValue 计算参数 n 的立方之后，并且在 cubeByValue 返回到 main 之前：



第 4 步：在 cubeByValue 返回到 main 之后，并且在把结果赋给 number 之前：



第 5 步：在 main 完成了对 number 赋值之后：

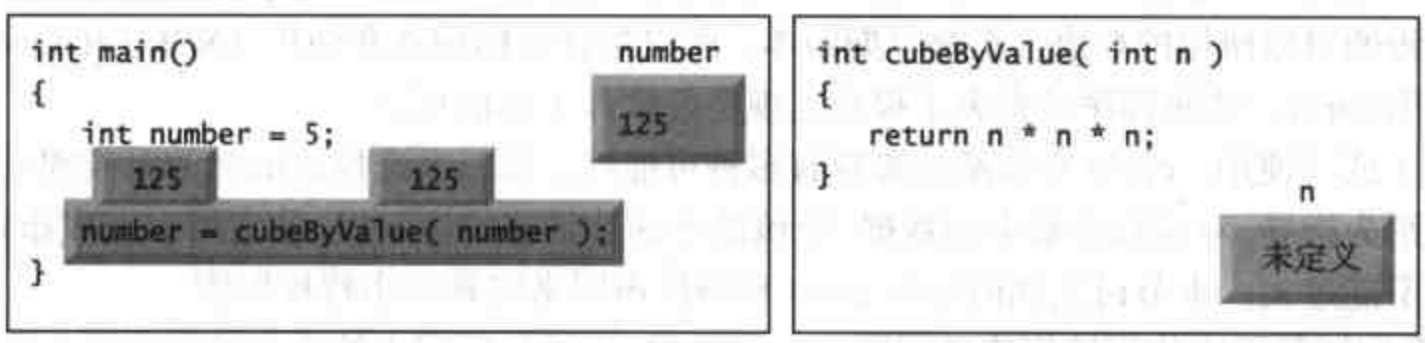
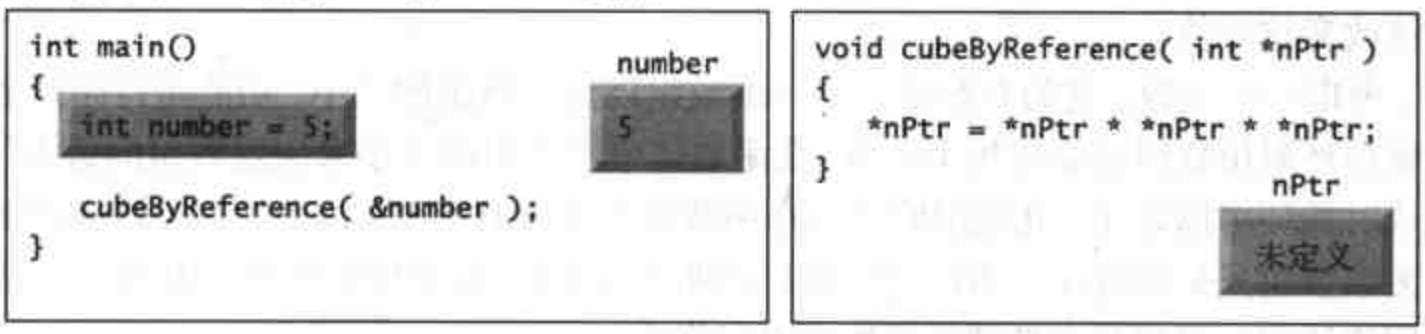
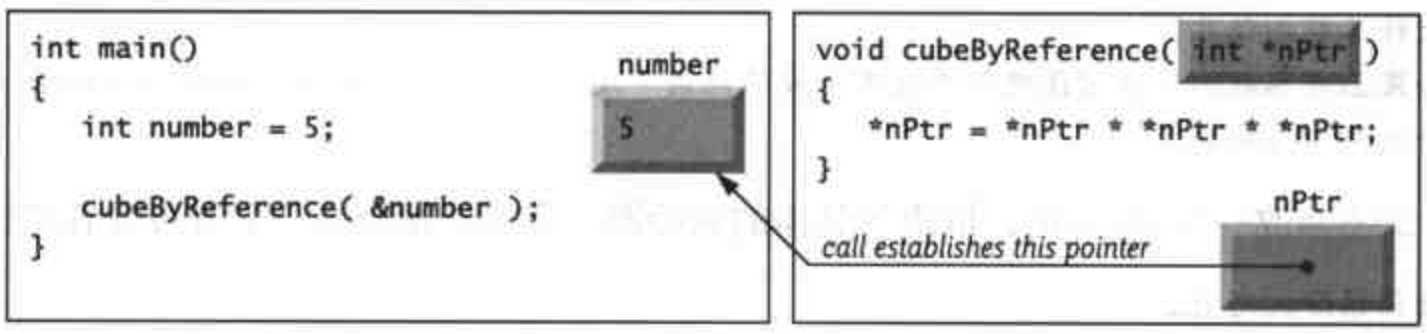


图 8.8　对图 8.6 中程序按值传递的分析（续）

第 1 步：在 main 调用 cubeByReference 之前：



第 2 步：在 cubeByReference 收到调用之后，并且在 *nPtr 被计算立方值之前：



第 3 步：在 *nPtr 被计算立方值之后，并且在程序控制返回到 main 之前：

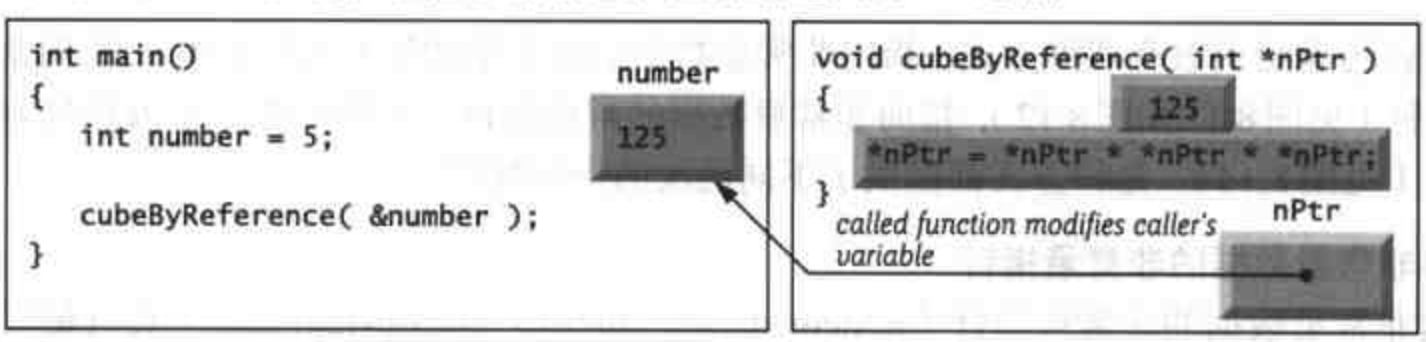


图 8.9　对图 8.7 中程序（用指针）按引用传递的分析

常量数据的非常量指针不需要包含const。可以用这样的指针在函数中接收以空字符终止的字符串，函数改变指针的值来处理（并且可能修改）字符串中的每个字符。回想一下，在7.4节中，一个以空字符终止的字符串可以存放到一个字符数组中，这个数组包含该字符串的字符和一个表示字符串结束的空字符。

```
1  // Fig. 8.10: fig08_10.cpp
2  // Converting lowercase letters to uppercase letters
3  // using a non-constant pointer to non-constant data.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include <cctype> // prototypes for islower and toupper
9  using std::islower;
10 using std::toupper;
11
12 void convertToUppercase( char * );
13
14 int main()
15 {
16    char phrase[] = "characters and $32.98";
17
18    cout << "The phrase before conversion is: " << phrase;
19    convertToUppercase( phrase );
20    cout << "\nThe phrase after conversion is:  " << phrase << endl;
21    return 0; // indicates successful termination
22 } // end main
23
24 // convert string to uppercase letters
25 void convertToUppercase( char *sPtr )
26 {
27    while ( *sPtr != '\0' ) // loop while current character is not '\0'
28    {
29       if ( islower( *sPtr ) ) // if character is lowercase,
30          *sPtr = toupper( *sPtr ); // convert to uppercase
31
32       sPtr++; // move sPtr to next character in string
33    } // end while
34 } // end function convertToUppercase
```

```
The phrase before conversion is: characters and $32.98
The phrase after conversion is:  CHARACTERS AND $32.98
```

图 8.10　把一个字符串转换为大写字母形式

在图 8.10 中，函数 convertToUppercase（第 25~34 行）声明参数 sPtr（第 25 行）为一个指向非常量数据的常量指针（没有用 const）。这个函数逐个处理存储在字符数组 phrase 中的以空字符终止的字符串中的字符（第 27~33 行）。别忘了一个字符数组的名字实际就是一个指向该数组第一个元素的指针，因此，可以把 phrase 作为参数传递给 convertToUppercase。函数 islower（第 29 行）接收一个字符参数，如果该字符是小写字符，则返回 true；否则返回 false。函数 toupper（第 30 行）接收一个字符参数——它把 ‘a’ 到 ‘z’ 之间的字符转换为相应的大写字母；其他的保持不变。如果字符是一个小写字母，则返回相应的大写字母；否则，返回原来的字符。函数 toupper 和函数 islower 是字符处理库<cctype>中的一部分。处理完一个字符之后，第 32 行将 sPtr 增加 1（如果 sPtr 声明为 const，这样就不行了）。当对指向数组的指针使用++运算符时，存储在该指针中的内存地址就会更改，指向数组的下一个元素（在这个例子中，是字符串中的下一个字符）。给指针加 1 是指针算术运算中的一个合法运算，将在 8.8 节和 8.9 节中详细介绍。

指向常量数据的非常量指针

指向常量数据的非常量指针（nonconstant pointer to constant data）可以被修改以指向任何适当类型的其他数据项，但是不能通过该指针来修改它所指向的数据。可以用这种指针为函数接收数组实参，函数处理数组的每个元素，但不允许修改数据。例如，函数printCharacters（图8.11中的第22~26行）声明参数sPtr（第22行）为const char*类型，这样，它可以接收一个以空字符终止的基于指针的字符串。这个声明从右到左读做“sPtr是指向一个字符常量的指针”。该函数的体用一条for语句（第24~25行）输出字符串的每个字符，直到遇到空字符为止。打印每个字符之后，指针sPtr进行自增以指向字符串中的下一个字符（该指针不是const的，所以这样是可以的）。main函数创建了字符数组phrase，将它传递给printCharacters。这里同样由于数组名实际是指向数组第一个元素的指针，因此我们可以将数组phrase传递给printCharacters。

```
1  // Fig. 8.11: fig08_11.cpp
2  // Printing a string one character at a time using
3  // a non-constant pointer to constant data.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  void printCharacters( const char * ); // print using pointer to const data
9
10 int main()
11 {
12    const char phrase[] = "print characters of a string";
13
14    cout << "The string is:\n";
15    printCharacters( phrase ); // print characters in phrase
16    cout << endl;
17    return 0; // indicates successful termination
18 } // end main
19
20 // sPtr can be modified, but it cannot modify the character to which
21 // it points, i.e., sPtr is a "read-only" pointer
22 void printCharacters( const char *sPtr )
23 {
24    for ( ; *sPtr != '\0'; sPtr++ ) // no initialization
25       cout << *sPtr; // display character without modification
26 } // end function printCharacters
```

```
The string is:
print characters of a string
```

图8.11 使用指向常量数据的非常量指针逐个字符地打印一个字符串

如果一个函数接收一个指向常量数据的非常量指针，然后试图用这个指针去修改数据。那么当编译这个函数时，会产生编译错误，如图8.12所示。注意：不同的编译器给出的编译错误信息可能会不同。

我们知道，数组是聚集的数据类型，将同类型的相关数据项存储在一个名字下。当用数组作为实参调用函数时，数组按引用传递给函数。但是，对象总是按值传递——整个对象的一个副本被传递。这需要运行期的开销来复制对象中的每个数据项，并将它存储到函数的调用堆栈。当对象必须要传递给函数时，我们可以用指向常量数据的指针（或对常量数据的引用）得到按引用传递的性能，同时得到按值传递的保护性。当传递指向对象的指针时，只会复制对象的地址；对象本身并没有被复制。在地址为四个字节的机器上，只是复制四个字节内存，而不需要复制可能庞大的对象。

```
1  // Fig. 8.12: fig08_12.cpp
2  // Attempting to modify data through a
3  // non-constant pointer to constant data.
4
5  void f( const int * ); // prototype
6
7  int main()
8  {
9     int y;
10
11    f( &y ); // f attempts illegal modification
12    return 0; // indicates successful termination
13 } // end main
14
15 // xPtr cannot modify the value of constant variable to which it points
16 void f( const int *xPtr )
17 {
18    *xPtr = 100; // error: cannot modify a const object
19 } // end function f
```

Borland C++ command-line compiler error message:

```
Error E2024 fig08_12.cpp 18:
   Cannot modify a const object in function f(const int *)
```

Microsoft Visual C++ compiler error message:

```
c:\scpphtp5_examples\ch08\Fig08_12\fig08_12.cpp(18) :
   error C2166: l-value specifies const object
```

GNU C++ compiler error message:

```
fig08_12.cpp: In function `void f(const int*)':
fig08_12.cpp:18: error: assignment of read-only location
```

图 8.12　试图通过指向常量数据的非常量指针修改数据



性能提示 8.1

如果大型对象不需要在被调用函数中修改，那么用指向常量数据的指针或对常量数据的引用来传递它们，可以获得按引用传递的性能。



软件工程知识 8.3

使用指向常量数据的指针或对常量数据的引用来传递大型对象，可以获得按值传递的安全性。

指向非常量数据的常量指针

指向非常量数据的常量指针（constant pointer to nonconstant data）始终指向同一个内存位置；通过该指针可以修改这个位置上的数据。这就是数组名的默认情况，数组名是指向这个数组开始处的常量指针。用数组名和数组下标，可以访问和修改数组中的所有元素。指向非常量数据的常量指针可以用来接收数组作为函数的实参，在函数中用数组下标表示法访问数组元素。声明为 const 的指针在声明时必须被初始化（如果指针是函数参数，它被传递给函数的指针初始化）。图 8.13 的程序试图修改一个常量指针。第 11 行声明指针 ptr 为 int* const 类型。这个声明从右到左读做“ptr 是指向一个非常量整数的常量指针”。该指针被初始化为整数变量 x 的地址。第 14 行试图把 y 的地址赋值给 ptr，但是编译器会产生一条错误消息。请注意，当第 13 行把 7 赋值给 *ptr 时不会发生错误——即使 ptr 本身被声明为 const，但是使用间接引用的 ptr 可以修改 ptr 所指向的非常量数据。

```
1  // Fig. 8.13: fig08_13.cpp
2  // Attempting to modify a constant pointer to non-constant data.
3
4  int main()
5  {
6     int x, y;
7
8     // ptr is a constant pointer to an integer that can
9     // be modified through ptr, but ptr always points to the
10    // same memory location.
11    int * const ptr = &x; // const pointer must be initialized
12
13    *ptr = 7; // allowed: *ptr is not const
14    ptr = &y; // error: ptr is const; cannot assign to it a new address
15    return 0; // indicates successful termination
16 } // end main
```

Borland C++ command-line compiler error message:

```
Error E2024 fig08_13.cpp 14: Cannot modify a const object in function main()s
```

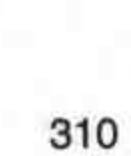
Microsoft Visual C++ compiler error message:

```
c:\scpphtp5_examples\ch08\Fig08_13\fig08_13.cpp(14) : error C2166:
   l-value specifies const object
```

GNU C++ compiler error message:

```
fig08_13.cpp: In function `int main()':
fig08_13.cpp:14: error: assignment of read-only variable `ptr'
```

图 8.13　试图修改指向非常量数据的常量指针



常见的编程错误 8.6

声明一个指针为 const 而不初始化它，是一个编译错误。

指向常量数据的常量指针

指向常量数据的常量指针（constant pointer to constant data）具有最小的访问权限。这种指针总是指向内存中相同的位置，并且不能用该指针修改这个内存位置的数据。如果一个函数接收一个数组做参数，只是用数组下标表示法读取数组，而不修改数组，那么应该用这种指针。图 8.14 的程序声明指针变量 ptr 为 const int* const 类型的（第 14 行）。这个声明从右到左读做“ptr 是指向一个整数常量的常量指针”。当试图修改 ptr 指向的数据（第 18 行）和存储在该指针变量中的地址（第 19 行）时，产生的错误信息如图所示。请注意，当程序试图间接引用 ptr，或当程序试图输出 ptr 所指向的值时（第 16 行），不会发生错误，因为在这个语句中既没有修改指针也没有修改它所指向的数据。

```
1  // Fig. 8.14: fig08_14.cpp
2  // Attempting to modify a constant pointer to constant data.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 5, y;
10
11    // ptr is a constant pointer to a constant integer.
```

图 8.14　试图修改指向常量数据的常量指针

```
12      // ptr always points to the same location; the integer
13      // at that location cannot be modified.
14      const int *const ptr = &x;
15
16      cout << *ptr << endl;
17
18      *ptr = 7; // error: *ptr is const; cannot assign new value
19      ptr = &y; // error: ptr is const; cannot assign new address
20      return 0; // indicates successful termination
21   } // end main
```

Borland C++ command-line compiler error message:

```
Error E2024 fig08_14.cpp 18: Cannot modify a const object in function main()
Error E2024 fig08_14.cpp 19: Cannot modify a const object in function main()
```

Microsoft Visual C++ compiler error message:

```
c:\scpphtp5_examples\ch08\Fig08_14\fig08_14.cpp(18) : error C2166:
   l-value specifies const object
c:\scpphtp5_examples\ch08\Fig08_14\fig08_14.cpp(19) : error C2166:
   l-value specifies const object
```

GNU C++ compiler error message:

```
fig08_14.cpp: In function `int main()':
fig08_14.cpp:18: error: assignment of read-only location
fig08_14.cpp:19: error: assignment of read-only variable `ptr'
```

图 8.14　试图修改指向常量数据的常量指针（续）

8.6　用按引用传递方式实现的选择排序

在这一节，我们定义了一个排序程序，示范按引用传递数组和单个的数组元素。我们采用选择排序（selection sort）算法，它是一个易于编写但效率较差的排序算法。算法的第一次迭代选择数组中最小的元素，然后把它和第一个元素交换。第二次迭代选择次小的元素（在剩余元素中是最小的元素），然后把它和第二个元素交换。算法一直进行，直到最后一次迭代选择出第二大的元素，并把它和倒数第二个元素交换，剩下最大的元素为最后一个元素。经过第i次迭代后，数组中最小的 i 个数据项按升序排放在数组的前 i 个元素中。

例如，考虑数组

```
34   56   4   10   77   51   93   30   5   52
```

一个实现选择排序的程序首先确定这个数组的最小元素（4），它在元素2中。程序把4和元素0（34）交换，得到

```
4   56   34   10   77   51   93   30   5   52
```

[注意：我们用粗体突出被交换的值。]然后，程序确定剩余元素（除了4的所有元素）中的最小值y它是5，包含在元素8中。程序把5和元素1（56）交换，得到

```
4   5   34   10   77   51   93   30   56   52
```

在第三次迭代中，程序确定下一个最小值（10），然后把它和元素2（34）交换。结果如下：

```
4   5   10   34   77   51   93   30   56   52
```

程序继续进行，直到数组全部排序完毕。

```
4   5   10   30   34   51   52   56   77   93
```

请注意，在第一次迭代之后，最小的元素位于第一个位置。在第二次迭代后，最小的两个元素依次位于前面两个位置。第三次迭代后，最小的三个元素依次位于前面三个位置。

图 8.15 利用两个函数 selectionSort 和 swap 实现选择排序。函数 selectionSort（第 36~53 行）对数组进行排序。第 38 行声明变量 smallest，它存储数组剩余元素中最小值的下标。第 41~52 行执行循环 size-1 次。第 43 行把最小元素的下标设置为当前下标。第 46~49 行循环遍历数组中的剩余元素。对其中的每个元素，第 48 行把它的值和最小元素的值相比较。如果当前元素小于最小元素，第 49 行把当前元素的下标赋给 smallest。当这个循环结束时，smallest 就包含了剩余数组元素中最小值的下标。第 51 行调用函数 swap（第 57~62 行）把剩余元素的最小值放到数组中的下一个位置（即把数组元素 array[i]和 array[smallest]进行交换）。

```
1  // Fig. 8.15: fig08_15.cpp
2  // This program puts values into an array, sorts the values into
3  // ascending order and prints the resulting array.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 void selectionSort( int * const, const int ); // prototype
12 void swap( int * const, int * const ); // prototype
13
14 int main()
15 {
16    const int arraySize = 10;
17    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19    cout << "Data items in original order\n";
20
21    for ( int i = 0; i < arraySize; i++ )
22       cout << setw( 4 ) << a[ i ];
23
24    selectionSort( a, arraySize ); // sort the array
25
26    cout << "\nData items in ascending order\n";
27
28    for ( int j = 0; j < arraySize; j++ )
29       cout << setw( 4 ) << a[ j ];
30
31    cout << endl;
32    return 0; // indicates successful termination
33 } // end main
34
35 // function to sort an array
36 void selectionSort( int * const array, const int size )
37 {
38    int smallest; // index of smallest element
39
40    // loop over size - 1 elements
41    for ( int i = 0; i < size - 1; i++ )
42    {
43       smallest = i; // first index of remaining array
```

图 8.15　采用按引用传递的选择排序

```
44
45       // loop to find index of smallest element
46       for ( int index = i + 1; index < size; index++ )
47
48          if ( array[ index ] < array[ smallest ] )
49             smallest = index;
50
51       swap( &array[ i ], &array[ smallest ] );
52    } // end if
53 } // end function selectionSort
54
55 // swap values at memory locations to which
56 // element1Ptr and element2Ptr point
57 void swap( int * const element1Ptr, int * const element2Ptr )
58 {
59    int hold = *element1Ptr;
60    *element1Ptr = *element2Ptr;
61    *element2Ptr = hold;
62 } // end function swap
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

图8.15　采用按引用传递的选择排序（续）

现在，我们进一步研究函数swap。C++在函数间进行信息隐藏，因此swap没有对selectionSort中单个数组元素的访问权限。因为selectionSort需要swap有访问数组元素的权限以交换元素，所以selectionSort按引用传递这些元素到swap中——明确地传递每个数组元素的地址。虽然整个数组是按引用传递的，但是单个的数组元素是标量，它们通常是按值传递的。因此，selectionSort在对swap的调用中给每个数组元素使用地址运算符&（第51行）来实现按引用传递。函数swap（第57~62行）将&array[i]接收到指针变量element1Ptr中。信息隐藏使swap不能“知道”名字array[i]，但swap可以用*element1Ptr，它作为array[i]的一个别名。这样，当swap引用*element1Ptr时，实际上在引用selectionSort中的array[i]。同样，当swap引用*element2Ptr时，它实际上引用了selectionSort中的array[smallest]。

尽管不允许swap使用下列的语句：

```
hold = array[ i ];
array[ i ] = array[ smallest ];
array[ smallest ] = hold;
```

但是，下面在图8.15的swap函数中的语句：

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

恰好达到同样的目的。

应注意函数selectionSort的几个特点。函数头部（第36行）声明array时，用int * const array来表示函数selectionSort接收一个一维数组作为参数，而不是用int array[]。array指针参数和size参数都声明为const，以实施最小特权原则。虽然参数size只是接收main函数中一个值的副本，修改这个副本不会改变main中的值，但是，selectionSort不需要改变size来完成它的任务——在selectionSort执行期间数组大小保持不变。因此，把size声明为const，确保它不会被修改。如果数组大小在排序过程中要修改，那么排序算法将不能正确地运行。

请注意，由于函数selectionSort必须知道数组的大小，才能排序数组，所以它接收数组大小作为一个参数。当基于指针的数组被传递到函数时，函数只是接收了该数组第一个元素的内存地址；数组大小必须单独传递给函数。

定义函数selectionSort接收数组大小作为一个参数之后，任何排序一维int数组的程序，对于任意的数组大小，都可以使用这个函数。数组的大小确定可以直接编写到函数中，但是，这将限制函数只能处理一个特定大小的数组，从而降低了函数的可重用性——只有处理此特定大小的一维int数组的程序才能使用这个函数。



软件工程知识 8.4
传递一个数组到函数时，也传递这个数组的大小（而不是在函数内部固定数组大小）。这样，可以使函数有更好的可重用性。

8.7 sizeof 运算符

C++ 提供一元运算符 sizeof，可确定程序编译期间数组（或任何其他数据类型、变量或常量）的字节大小。当sizeof运算符应用于一个数组名时，如图8.16所示（第14行），它返回该数组的总字节数，返回值是size_t类型的（在多数编译器中是unsigned int的一个别名）。注意，这和一个vector<int>的大小是不同的，后者是vector中的整数元素的数目。我们用来编译这个程序的计算机用8个字节的内存存储一个double类型的变量，array声明有20个元素（第12行），因此，array使用了内存中的160个字节。当sizeof运算符应用于函数中接收数组作为实参的指针参数时（第24行），它返回这个指针的字节数（4），而不是该数组的大小。

```
1  // Fig. 8.16: fig08_16.cpp
2  // Sizeof operator when used on an array name
3  // returns the number of bytes in the array.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  size_t getSize( double * ); // prototype
9
10 int main()
11 {
12    double array[ 20 ]; // 20 doubles; occupies 160 bytes on our system
13
14    cout << "The number of bytes in the array is " << sizeof( array );
15
16    cout << "\nThe number of bytes returned by getSize is "
17       << getSize( array ) << endl;
18    return 0; // indicates successful termination
19 } // end main
20
21 // return size of ptr
22 size_t getSize( double *ptr )
23 {
24    return sizeof( ptr );
25 } // end function getSize
```

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

图 8.16 对一个数组名使用 sizeof 运算符时，返回该数组中的字节数



常见的编程错误 8.7

在一个函数中，使用sizeof运算符来获取一个数组参数的字节数，结果会得到一个指针的字节数，而不是数组的字节数。

注意：当用 Borland C++ 编译器编译图 8.16 的程序时，编译器产生警告消息 "Parameter 'ptr' is never used in function getSize(double *)."。产生警告的原因是，sizeof 事实上是一个编译时运算符，因此，函数体在执行时不会用到变量 ptr。很多编译器都会产生诸如此类的警告消息，让你知道变量没有被使用，这样，你可以从代码中删除它，或者修改代码以正确地使用该变量。在图 8.17 中就不同的编译器而言也会产生类似的消息。

```
// Fig. 8.17: fig08_17.cpp
// Demonstrating the sizeof operator.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
   char c; // variable of type char
   short s; // variable of type short
   int i; // variable of type int
   long l; // variable of type long
   float f; // variable of type float
   double d; // variable of type double
   long double ld; // variable of type long double
   int array[ 20 ]; // array of int
   int *ptr = array; // variable of type int *

   cout << "sizeof c = " << sizeof c
      << "\tsizeof(char) = " << sizeof( char )
      << "\nsizeof s = " << sizeof s
      << "\tsizeof(short) = " << sizeof( short )
      << "\nsizeof i = " << sizeof i
      << "\tsizeof(int) = " << sizeof( int )
      << "\nsizeof l = " << sizeof l
      << "\tsizeof(long) = " << sizeof( long )
      << "\nsizeof f = " << sizeof f
      << "\tsizeof(float) = " << sizeof( float )
      << "\nsizeof d = " << sizeof d
      << "\tsizeof(double) = " << sizeof( double )
      << "\nsizeof ld = " << sizeof ld
      << "\tsizeof(long double) = " << sizeof( long double )
      << "\nsizeof array = " << sizeof array
      << "\nsizeof ptr = " << sizeof ptr << endl;
   return 0; // indicates successful termination
} // end main
```

```
sizeof c = 1     sizeof(char) = 1
sizeof s = 2     sizeof(short) = 2
sizeof i = 4     sizeof(int) = 4
sizeof l = 4     sizeof(long) = 4
sizeof f = 4     sizeof(float) = 4
sizeof d = 8     sizeof(double) = 8
sizeof ld = 8    sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4
```

图 8.17　使用 sizeof 运算符确定基本数据类型的大小



可移植性提示 8.4

现在大多数计算机使用 2 个字节或 4 个字节的整数。一些新的计算机使用 8 个字节的整数。由于指针算术运算的结果取决于指针所指向的对象的大小，所以指针算术运算是与机器相关的。

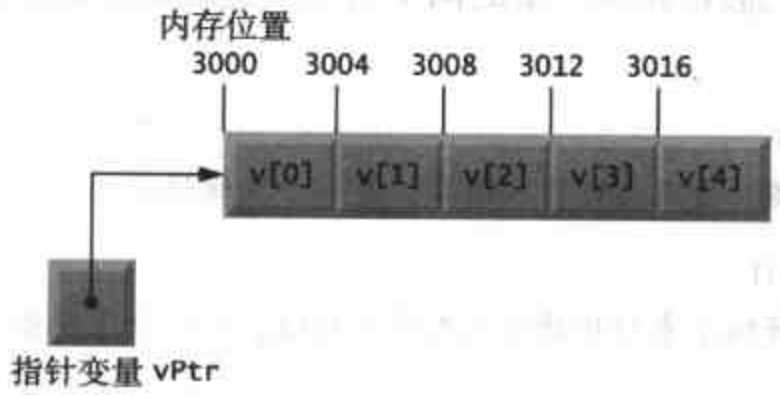


图 8.18　数组 v 和指向数组 v 的指针变量 vPtr

在常规的算术中，3000+2 得到值 3002。通常，指针算术并不是这样的。当一个指针加上或减去一个整数时，不是简单地加上或减去这个整数，而是加上或减去这个整数与该指针指向对象的大小的乘积。字节数取决于对象的数据类型。例如，下面的语句：

```
vPtr += 2 ;
```

会得到 3008（3000+2*4），假设一个 int 数据占 4 个字节内存。此时，在数组 v 中，vPtr 指向 v[2]（见图 8.19）。如果一个整数占 2 个字节内存，那么前面的计算将得到内存地址 3004（3000+2*2）。如果数组为不同的数据类型，前面的语句将使指针增加其所指向的数据类型对象的字节数的两倍。对字符数组进行指针运算时，结果和平常的算术计算一样，因为每个字符的长度就是一个字节。

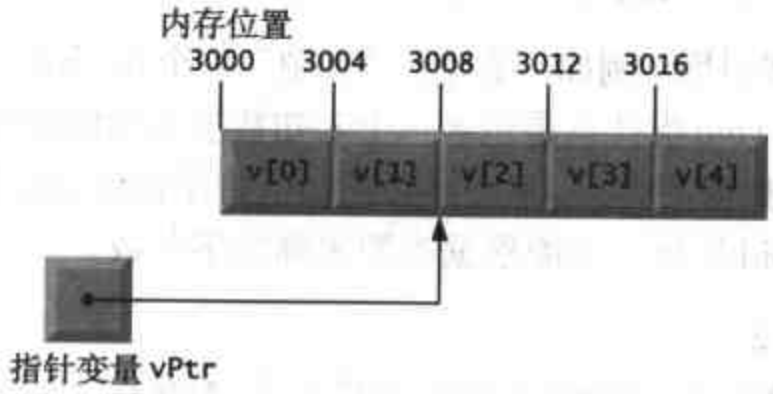


图 8.19　指针算术运算之后的指针 vPtr

如果 vPtr 增加到了 3016，即指向 v[4]，那么如下语句：

```
vPtr -= 4 ;
```

将使 vPtr 退回到 3000，也就是数组的开始地址。如果指针加 1 或减 1，则可以使用自增（++）和自减（--）运算符。下列语句：

```
++vPtr;
vPtr++;
```

的每一条都是将指针加 1，使它指向数组的下一个元素。下列语句：

```
--vPtr;
vPtr--;
```

的每一条都将指针减 1，使它指向数组的前一个元素。

指向同一个数组的指针变量可以相减。例如，如果 vPtr 包含地址 3000，v2Ptr 包含地址 3008，那么下述语句：

```
1  // Fig. 8.20: fig08_20.cpp
2  // Using subscripting and pointer notations with arrays.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int b[] = { 10, 20, 30, 40 }; // create 4-element array b
10    int *bPtr = b; // set bPtr to point to array b
11
12    // output array b using array subscript notation
13    cout << "Array b printed with:\n\nArray subscript notation\n";
14
15    for ( int i = 0; i < 4; i++ )
16       cout << "b[" << i << "] = " << b[ i ] << '\n';
17
18    // output array b using the array name and pointer/offset notation
19    cout << "\nPointer/offset notation where "
20       << "the pointer is the array name\n";
21
22    for ( int offset1 = 0; offset1 < 4; offset1++ )
23       cout << "*(b + " << offset1 << ") = " << *( b + offset1 ) << '\n';
24
25    // output array b using bPtr and array subscript notation
26    cout << "\nPointer subscript notation\n";
27
28    for ( int j = 0; j < 4; j++ )
29       cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
30
31    cout << "\nPointer/offset notation\n";
32
33    // output array b using bPtr and pointer/offset notation
34    for ( int offset2 = 0; offset2 < 4; offset2++ )
35       cout << "*(bPtr + " << offset2 << ") = "
36          << *( bPtr + offset2 ) << '\n';
37
38    return 0; // indicates successful termination
39 } // end main
```

```
Array b printed with:

Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

图 8.20　用数组名和指针引用数组元素

为了进一步说明数组和指针的可交换性，让我们看看图8.21中的两个字符串复制函数——copy1和copy2。这两个函数都把一个字符串复制到一个字符数组中。比较copy1和copy2的函数原型，发现这两个函数看起来是一样的（由于数组和指针的可交换性）。两个函数完成相同的任务，但它们的实现是不同的。

```
1  // Fig. 8.21: fig08_21.cpp
2  // Copying a string using array notation and pointer notation.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void copy1( char *, const char * ); // prototype
8  void copy2( char *, const char * ); // prototype
9
10 int main()
11 {
12    char string1[ 10 ];
13    char *string2 = "Hello";
14    char string3[ 10 ];
15    char string4[] = "Good Bye";
16
17    copy1( string1, string2 ); // copy string2 into string1
18    cout << "string1 = " << string1 << endl;
19
20    copy2( string3, string4 ); // copy string4 into string3
21    cout << "string3 = " << string3 << endl;
22    return 0; // indicates successful termination
23 } // end main
24
25 // copy s2 to s1 using array notation
26 void copy1( char * s1, const char * s2 )
27 {
28    // copying occurs in the for header
29    for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )
30       ; // do nothing in body
31 } // end function copy1
32
33 // copy s2 to s1 using pointer notation
34 void copy2( char *s1, const char *s2 )
35 {
36    // copying occurs in the for header
37    for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
38       ; // do nothing in body
39 } // end function copy2
```

```
string1 = Hello
string3 = Good Bye
```

图8.21　使用数组表示法和指针表示法的字符串复制

函数copy1（第26~31行）使用数组下标表示法把s2中的字符串复制到字符数组s1中。该函数声明了一个整数计数器变量i，用来作为数组下标。for语句的头部（第29行）进行了全部的复制操作——它的循环体是空语句。该头部指定i初始化为0，并在循环的每次迭代时增加1。for语句的条件：(s1[i] = s2[i])!= '\0'，执行从s2到s1逐个字符的复制操作。当在s2中遇到空字符时，将它赋值给s1，然后循环终止，因为空字符等于'\0'。请记住，一条赋值语句的值是赋给它的左操作数的值。

函数copy2（第34~39行）使用指针和指针算术运算把s2中的字符串复制到字符数组s1。同样，for语句的头部（第37行）进行了全部的复制操作。该头部没有包含任何变量的初始化。和函数copy1中一样，条件(*s1 = *s2)!= '\0'执行复制操作。指针s2被间接引用，把得到的字符赋给间接引用的指

我们用自顶向下、逐步求精的方法开发一个程序，它可以洗52张牌，然后发出52张牌。自顶向下的方法尤其适用于解决比我们之前所看到的更大型、更复杂的问题。

我们用一个4 × 13的二维数组表示一副要玩的牌，如图8.23所示。行对应花色——第0行对应红桃（Hearts），第1行对应方块（Diamonds），第2行对应梅花（clubs），第3行对应黑桃（Spades）。列对应牌的面值——第0列到第9列分别对应1到10，第10列到第12列分别对应J、Q和K。我们将装入字符串数组suit和face，其中suit用字符串表示4种花色（如图8.22所示），face用字符串表示13张牌。

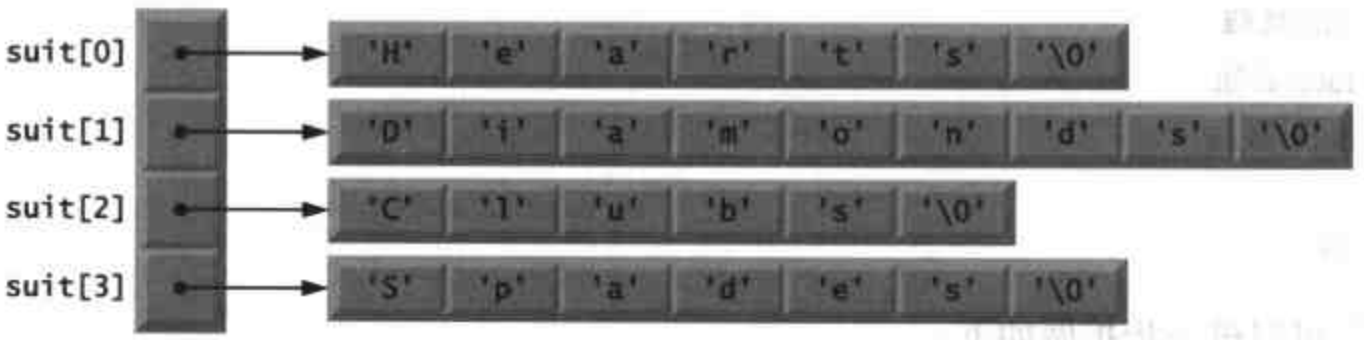


图 8.22　suit 数组的图形化表示

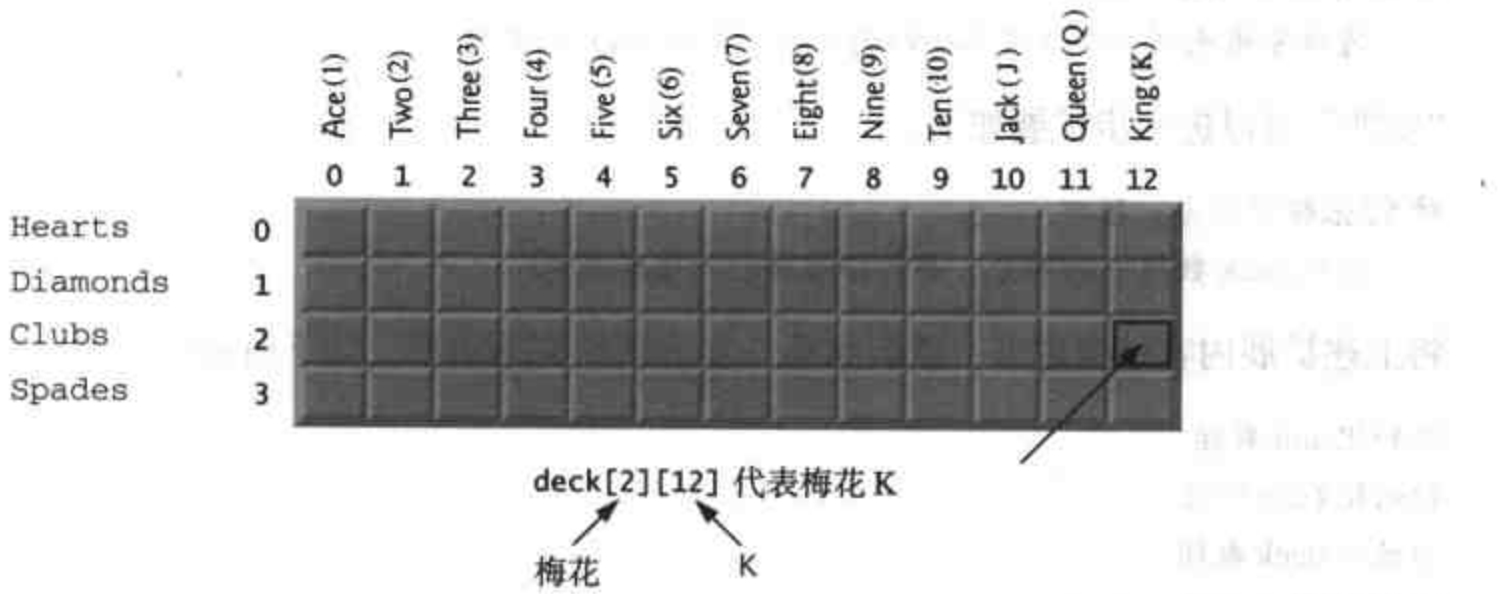


图 8.23　一副牌的二维数组表示

这副模拟的牌可以按以下操作进行洗牌。首先，把数组deck初始化为0。接着分别随机选择row（0到3）和column（0到12）。在数组元素deck[row][column]中填充数字1，表示这张牌是洗牌完成后将要发的第一张牌。这个过程一直持续下去，把数字2、3、……、52随机填充到数组deck中，表示在完成洗牌后的牌中哪些牌放在第2张、第3张、……、第52张发出。当数组deck开始用牌号进行填充时，一张牌可能被第二次选中（即当deck[row][column]被选中时其值是非零的）。这时忽略这个选择，并继续随机选择其他行和列，直到找到一张未选择过的牌。最后，从1到52的数值将被填充到deck数组的52个元素中。此时，这副牌已经完全洗好了。

如果已经洗过的牌被重复地随机选择，那么这种洗牌算法可能要执行无限长的时间。这种现象就是所谓的无限延迟（indefinite postponement），也称为饿死（starvation）。在习题中，我们将讨论一个更好的、排除了无限延迟可能性的洗牌算法。



性能提示 8.3

有时，一些看似“自然”的算法可能会包含诸如无限延迟这样的微妙的性能问题。因此，需要寻找可以避免无限延迟的算法。

要发第1张牌，我们要在数组中查找匹配1的元素deck[row][column]。这可以用一条嵌套的for语句来完成，row取0到3，column取0到12。这个数组元素对应的是哪张牌呢？suit数组已经预先装入了4种花色，因此，为了得到花色，我们只要打印字符串suit[row]。类似地，为了得到牌的

deal（图 8.26 的第 58~88 行）实现了图 8.24 的第 13~16 行。请注意，函数 deal 中使用的输出格式（图 8.26 的第 81~83 行）。该输出语句在 5 个字符的域宽中按向右对齐输出面值，在 8 个字符的域宽中按向左对齐输出花色（见图 8.27）。输出打印成两列，如果当前输出的牌在第 1 列，则在后面输出一个制表符，移到第 2 列（第 83 行）；否则，输出换行符。

```
初始化 suit 数组
初始化 face 数组
初始化 deck 数组

对 52 张牌中的每张牌
    随机选择 deck 的一个元素

    当所选的 deck 元素之前已经被选择过时
        随机选择一个 deck 元素

    将牌号填充到所选择的 deck 数组元素中

对 52 张牌中的每张牌
    对 deck 数组的每个元素
        如果该元素包含需要的牌号
            打印该牌的面值和花色
```

图 8.24　洗牌和发牌程序的伪代码算法

```
// Fig. 8.25: DeckOfCards.h
// Definition of class DeckOfCards that
// represents a deck of playing cards.

// DeckOfCards class definition
class DeckOfCards
{
public:
   DeckOfCards(); // constructor initializes deck
   void shuffle(); // shuffles cards in deck
   void deal(); // deals cards in deck
private:
   int deck[ 4 ][ 13 ]; // represents deck of cards
}; // end class DeckOfCards
```

图 8.25　DeckOfCards 的头文件

```
// Fig. 8.26: DeckOfCards.cpp
// Member-function definitions for class DeckOfCards that simulates
// the shuffling and dealing of a deck of playing cards.
#include <iostream>
using std::cout;
using std::left;
using std::right;

#include <iomanip>
using std::setw;

#include <cstdlib> // prototypes for rand and srand
using std::rand;
using std::srand;

#include <ctime> // prototype for time
```

图 8.26　洗牌和发牌所需成员函数的定义

```
17 using std::time;
18
19 #include "DeckOfCards.h" // DeckOfCards class definition
20
21 // DeckOfCards default constructor initializes deck
22 DeckOfCards::DeckOfCards()
23 {
24    // loop through rows of deck
25    for ( int row = 0; row <= 3; row++ )
26    {
27       // loop through columns of deck for current row
28       for ( int column = 0; column <= 12; column++ )
29       {
30          deck[ row ][ column ] = 0; // initialize slot of deck to 0
31       } // end inner for
32    } // end outer for
33
34    srand( time( 0 ) ); // seed random number generator
35 } // end DeckOfCards default constructor
36
37 // shuffle cards in deck
38 void DeckOfCards::shuffle()
39 {
40    int row; // represents suit value of card
41    int column; // represents face value of card
42
43    // for each of the 52 cards, choose a slot of the deck randomly
44    for ( int card = 1; card <= 52; card++ )
45    {
46       do // choose a new random location until unoccupied slot is found
47       {
48          row = rand() % 4; // randomly select the row
49          column = rand() % 13; // randomly select the column
50       } while( deck[ row ][ column ] != 0 ); // end do...while
51
52       // place card number in chosen slot of deck
53       deck[ row ][ column ] = card;
54    } // end for
55 } // end function shuffle
56
57 // deal cards in deck
58 void DeckOfCards::deal()
59 {
60    // initialize suit array
61    static const char *suit[ 4 ] =
62       { "Hearts", "Diamonds", "Clubs", "Spades" };
63
64    // initialize face array
65    static const char *face[ 13 ] =
66       { "Ace", "Deuce", "Three", "Four", "Five", "Six", "Seven",
67       "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
68
69    // for each of the 52 cards
70    for ( int card = 1; card <= 52; card++ )
71    {
72       // loop through rows of deck
73       for ( int row = 0; row <= 3; row++ )
74       {
75          // loop through columns of deck for current row
76          for ( int column = 0; column <= 12; column++ )
77          {
78             // if slot contains current card, display card
79             if ( deck[ row ][ column ] == card )
```

图 8.26　洗牌和发牌所需成员函数的定义（续）

```
            {
               cout << setw( 5 ) << right << face[ column ]
                  << " of " << setw( 8 ) << left << suit[ row ]
                  << ( card % 2 == 0 ? '\n' : '\t' );
            } // end if
         } // end innermost for
      } // end inner for
   } // end outer for
} // end function deal
```

图 8.26　洗牌和发牌所需成员函数的定义（续）

```
// Fig. 8.27: fig08_27.cpp
// Card shuffling and dealing program.
#include "DeckOfCards.h" // DeckOfCards class definition

int main()
{
   DeckOfCards deckOfCards; // create DeckOfCards object

   deckOfCards.shuffle(); // shuffle the cards in the deck
   deckOfCards.deal(); // deal the cards in the deck
   return 0; // indicates successful termination
} // end main
```

```
 Nine of Spades         Seven of Clubs
 Five of Spades         Eight of Clubs
Queen of Diamonds       Three of Hearts
 Jack of Spades          Five of Diamonds
 Jack of Diamonds       Three of Diamonds
Three of Clubs            Six of Clubs
  Ten of Clubs           Nine of Diamonds
  Ace of Hearts         Queen of Hearts
Seven of Spades         Deuce of Spades
  Six of Hearts         Deuce of Clubs
  Ace of Clubs          Deuce of Diamonds
 Nine of Hearts         Seven of Diamonds
  Six of Spades         Eight of Diamonds
  Ten of Spades          King of Hearts
 Four of Clubs            Ace of Spades
  Ten of Hearts          Four of Spades
Eight of Hearts         Eight of Spades
 Jack of Hearts           Ten of Diamonds
 Four of Diamonds        King of Diamonds
Seven of Hearts          King of Spades
Queen of Spades          Four of Hearts
 Nine of Clubs            Six of Diamonds
Deuce of Hearts          Jack of Clubs
 King of Clubs          Three of Spades
Queen of Clubs           Five of Clubs
 Five of Hearts           Ace of Diamonds
```

图 8.27　洗牌和发牌程序

这个发牌程序也有一个缺点。一旦找到匹配元素，即使是在第一次尝试中找到的，内部的两条 for 语句仍然会继续查找 deck 中的剩余元素。在习题中，我们将弥补这个缺陷。

8.12　函数指针

指向函数的指针包含内存中该函数的地址。在第7章，我们知道数组名实际上是数组第一个元素在内存中的地址。类似地，函数名实际上是执行这个函数任务的代码在内存中的开始地址。指向函数的指针可以传递给函数、从函数返回、存储在数组中以及赋给其他的函数指针。

使用函数指针的多用途选择排序

为了阐明函数指针的用法，我们把图8.15的选择排序程序进行了修改，如图8.28所示。图8.28中的程序由main函数（第17~55行）和函数selectionSort（第59~76行）、swap（第80~85行）、ascending（第89~92行）以及descending（第96~99行）等组成。函数selectionSort除了接收一个要排序的数组和该数组的大小作为参数之外，还接收一个函数指针作为参数，这个函数指针可以指向函数ascending或者指向函数descending。函数descending和ascending决定排序的顺序。程序提示用户选择是按升序还是按降序对数组进行排序（第24~26行）。如果用户输入1，则传递一个指向函数ascending的指针给selectionSort（第37行），使数组按升序进行排序。如果用户输入2，则传递一个指向函数descending的指针给selectionSort（第45行），使数组按降序进行排序。

```
1  // Fig. 8.28: fig08_28.cpp
2  // Multipurpose sorting program using function pointers.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 // prototypes
12 void selectionSort( int [], const int, bool (*)( int, int ) );
13 void swap( int * const, int * const );
14 bool ascending( int, int ); // implements ascending order
15 bool descending( int, int ); // implements descending order
16
17 int main()
18 {
19    const int arraySize = 10;
20    int order; // 1 = ascending, 2 = descending
21    int counter; // array index
22    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
23
24    cout << "Enter 1 to sort in ascending order,\n"
25       << "Enter 2 to sort in descending order: ";
26    cin >> order;
27    cout << "\nData items in original order\n";
28
29    // output original array
30    for ( counter = 0; counter < arraySize; counter++ )
31       cout << setw( 4 ) << a[ counter ];
32
33    // sort array in ascending order; pass function ascending
34    // as an argument to specify ascending sorting order
35    if ( order == 1 )
36    {
37       selectionSort( a, arraySize, ascending );
38       cout << "\nData items in ascending order\n";
39    } // end if
40
41    // sort array in descending order; pass function descending
42    // as an argument to specify descending sorting order
43    else
44    {
45       selectionSort( a, arraySize, descending );
46       cout << "\nData items in descending order\n";
47    } // end else part of if...else
```

图8.28 使用函数指针的多用途排序程序

```
48
49    // output sorted array
50    for ( counter = 0; counter < arraySize; counter++ )
51       cout << setw( 4 ) << a[ counter ];
52
53    cout << endl;
54    return 0; // indicates successful termination
55 } // end main
56
57 // multipurpose selection sort; the parameter compare is a pointer to
58 // the comparison function that determines the sorting order
59 void selectionSort( int work[], const int size,
60                     bool (*compare)( int, int ) )
61 {
62    int smallestOrLargest; // index of smallest (or largest) element
63
64    // loop over size - 1 elements
65    for ( int i = 0; i < size - 1; i++ )
66    {
67       smallestOrLargest = i; // first index of remaining vector
68
69       // loop to find index of smallest (or largest) element
70       for ( int index = i + 1; index < size; index++ )
71          if ( !(*compare)( work[ smallestOrLargest ], work[ index ] ) )
72             smallestOrLargest = index;
73
74       swap( &work[ smallestOrLargest ], &work[ i ] );
75    } // end if
76 } // end function selectionSort
77
78 // swap values at memory locations to which
79 // element1Ptr and element2Ptr point
80 void swap( int * const element1Ptr, int * const element2Ptr )
81 {
82    int hold = *element1Ptr;
83    *element1Ptr = *element2Ptr;
84    *element2Ptr = hold;
85 } // end function swap
86
87 // determine whether element a is less than
88 // element b for an ascending order sort
89 bool ascending( int a, int b )
90 {
91    return a < b; // returns true if a is less than b
92 } // end function ascending
93
94 // determine whether element a is greater than
95 // element b for a descending order sort
96 bool descending( int a, int b )
97 {
98    return a > b; // returns true if a is greater than b
99 } // end function descending
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

图 8.28　使用函数指针的多用途排序程序（续）

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in descending order
  89  68  45  37  12  10   8   6   4   2
```

图 8.28 使用函数指针的多用途排序程序（续）

在第 60 行 selectionSort 的函数头部有如下参数：

```
bool ( *compare )( int, int )
```

这个参数指定了一个函数指针。关键字bool表示所指向的函数返回一个bool值。文本(*compare)表示函数指针的名字（此处的 * 表明参数 compare 是一个指针）。文本(int, int)表示 compare 所指向的函数有两个整数参数。*compare的两边要加上圆括号，才能表示compare是一个函数指针。如果不加圆括号，该声明将变成下述形式：

```
bool *compare( int, int )
```

那么它表示了一个接收两个整数做参数、并返回一个指向 bool 值的指针的函数。

在 selectionSort 的函数原型中，相应的参数为：

```
bool (*)( int, int )
```

请注意这里只包含了类型。为了加强程序的文档性，程序员可以加上变量名，但编译器会忽略它们。

在第 71 行，传递给 selectionSort 的函数以如下形式调用：

```
( *compare )( work[ smallestOrLargest ], work[ index ] )
```

就像间接引用变量指针可以访问变量值一样，间接引用函数指针可以执行该函数。*compare两边的圆括号是必需的，如果没有，*运算符将试图间接引用函数调用所返回的值。不间接引用函数指针，也可以调用该函数，如下所示：

```
compare( work[ smallestOrLargest ], work[ index ] )
```

这里将指针直接作为函数名使用。我们提倡通过指针调用函数的第一种方法，因为它显式地说明compare是一个函数指针，通过间接引用这个指针来调用函数。通过指针调用函数的第二种方法使compare 看起来好像是程序中的一个实际函数的名字。这会使程序的用户感到迷惑，他想看函数compare 的定义，但却发现在文件中它并没有被定义。

函数指针数组

函数指针的一个用途是用在菜单驱动的系统中。例如，一个程序可能会提示用户输入一个整数值，从菜单中选择一个选项。可以把用户的选择作为一个函数指针数组的下标，数组中的指针用来调用这个函数。

图 8.29 提供了一个声明和使用函数指针数组的例子。该程序定义了 3 个函数——function0、function1 和 function2，每个函数都有一个整数参数，且不返回任何值。第 17 行将指向这 3 个函数的指针存储到数组 f 中。在这种情况下，数组所指向的所有函数必须具有相同的返回值和相同的参数类型。第 17 行的声明从最左边的括号开始读做“f 是一个有 3 个函数指针的数组，每个指针所指

向的函数都有一个 int 参数并返回 void”。该数组用 3 个函数名（也是指针）进行了初始化。程序提示用户输入 0 到 2 之间的数字，或输入 3 结束。当用户输入一个 0 到 2 之间的值时，用这个值作为函数指针数组的下标。第 29 行调用数组 f 中的一个相应函数。调用时，f[choice]选择数组中 choice 位置的指针。间接引用该指针来调用函数，并把 choice 作为参数传递给函数。每个函数打印它的参数值和函数名，表示该函数被正确地调用。在习题中，你将开发一个菜单驱动的系统。我们在第 13 章中将会看到，编译器开发者使用函数指针数组来实现支持虚函数（virtual function）的机制，而虚函数是支持多态性的关键技术。

```
1  // Fig. 8.29: fig08_29.cpp
2  // Demonstrating an array of pointers to functions.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  // function prototypes -- each function performs similar actions
9  void function0( int );
10 void function1( int );
11 void function2( int );
12
13 int main()
14 {
15    // initialize array of 3 pointers to functions that each
16    // take an int argument and return void
17    void (*f[ 3 ])( int ) = { function0, function1, function2 };
18
19    int choice;
20
21    cout << "Enter a number between 0 and 2, 3 to end: ";
22    cin >> choice;
23
24    // process user's choice
25    while ( ( choice >= 0 ) && ( choice < 3 ) )
26    {
27       // invoke the function at location choice in
28       // the array f and pass choice as an argument
29       (*f[ choice ])( choice );
30
31       cout << "Enter a number between 0 and 2, 3 to end: ";
32       cin >> choice;
33    } // end while
34
35    cout << "Program execution completed." << endl;
36    return 0; // indicates successful termination
37 } // end main
38
39 void function0( int a )
40 {
41    cout << "You entered " << a << " so function0 was called\n\n";
42 } // end function function0
43
44 void function1( int b )
45 {
46    cout << "You entered " << b << " so function1 was called\n\n";
47 } // end function function1
48
49 void function2( int c )
50 {
```

图 8.29　函数指针数组

```
51     cout << "You entered " << c << " so function2 was called\n\n";
52  } // end function function2
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function0 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function1 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function2 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

图 8.29　函数指针数组（续）

8.13　基于指针的字符串处理概述

在这一节，我们介绍一些通用的 C++ 标准库函数，它们使字符串处理更加方便。这里讨论的技术适用于开发文本编辑器、字处理器、版面设计软件、计算机排版系统和其他的文本处理软件。我们已经在几个例子中使用过 C++ 标准库 string 类，将字符串表示为成熟的对象。例如，在第 3 至第 7 章的类 GradeBook 的实例研究中，就用 string 对象表示课程名称。虽然使用 string 对象比较简单，我们在这一节还是使用以空字符结尾的、基于指针的字符串。许多C++标准库函数只对以空字符结尾的、基于指针的字符串起作用，而使用这种指针会比使用 string 对象复杂。同样，如果要处理遗留的 C++ 程序，就需要操作这些基于指针的字符串。

8.13.1　字符和基于指针的字符串基础

字符是 C++ 源程序的基本组成部分。每个程序都是由一系列字符构成的，这些字符组合在一起才有意义，而编译器把它们解释为一系列用来完成任务的指令。程序可以包含字符常量。一个字符常量就是一个整数值，表示为用单引号引起来的一个字符。字符常量的值是机器字符集中该字符的整数值。例如，'z' 表示 z 的整数值（在 ASCII 字符集中为 122，参见附录 B），'\n' 表示换行符的整数值（在 ASCII 字符集中为 10）。

字符串是被当做一个单元来处理的一系列字符。字符串可以包含字符、数组和各种特殊字符，例如 +、–、*、/ 和 $。在 C++ 中，文字串或字符串常量都放在双引号中，如下所示：

```
"John Q. Doe"                (一个名字)
"9999 Main Street"           (一个街道地址)
"Maynard, Massachusetts"     (一个城市和一个州)
"(201)555-1212"              (一个电话号码)
```

在 C++ 中，基于指针的字符串是以空字符（'\0'）结尾的字符数组，这个空字符标记了字符串在内存中结束的位置。通过指向字符串第一个字符的指针访问该字符串。一个字符串的值就是它的第一个字符的地址。因此，在 C++ 中，更恰当的说法是“字符串是一个常量指针”，实际上也就是指向该字符串第一个字符的指针。从这个意义上讲，字符串和数组是类似的，因为数组名也是一个指向它第一个元素的指针。

无论是在字符数组的声明中，还是在 char* 类型的变量的声明中，都可以将文字串作为初始化值。下列声明

用 strcpy 和 strncyp 复制字符串

函数 strcpy 将它的第二个参数（一个字符串）复制到它的第一个参数（一个字符数组）中，该字符数组必须足够大，以便存储字符串和同样被复制的终止空字符。函数 strncpy 除了要指定从字符串复制到数组的字符数之外，它和 strcpy 是一样的。请注意，函数 strncpy 不需要复制它的第二个参数的终止空字符，只有当要复制的字符数至少比字符串的长度大 1 时才复制终止空字符。例如，如果 "test" 是第二个参数，那么只有当 strncpy 的第三个参数至少为 5（"test" 中的 4 个字符加上一个终止空字符）时，才会写入一个终止空字符。如果第三个参数比 5 大，就会在数组中一直添加终止空字符，直到写入的总字符数达到第三个参数所指定的数值。



常见的编程错误 8.20

当使用 strncpy 时，如果 strncpy 第三个参数所指定要复制的字符数不大于第二个参数的长度，那么第二个参数（一个 char* 字符串）的终止空字符就不会被复制。在这样的情况下，如果程序员不用一个空字符人为地终止得到的 char* 字符串，就会导致一个致命的错误。

图 8.31 使用 strcpy（第 17 行）把数组 x 中的整个字符串复制到数组 y 中，而使用 strncpy（第 23 行）把数组 x 中的前 14 个字符复制到数组 z 中。第 24 行追加一个空字符（'\0'）到数组 z，因为程序中对 strncpy 的调用并没有写入一个终止空字符（第三个参数比第二个参数的字符串字面长度加 1 要小）。

```
1  // Fig. 8.31: fig08_31.cpp
2  // Using strcpy and strncpy.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // prototypes for strcpy and strncpy
8  using std::strcpy;
9  using std::strncpy;
10
11 int main()
12 {
13    char x[] = "Happy Birthday to You"; // string length 21
14    char y[ 25 ];
15    char z[ 15 ];
16
17    strcpy( y, x ); // copy contents of x into y
18
19    cout << "The string in array x is: " << x
20       << "\nThe string in array y is: " << y << '\n';
21
22    // copy first 14 characters of x into z
23    strncpy( z, x, 14 ); // does not copy null character
24    z[ 14 ] = '\0'; // append '\0' to z's contents
25
26    cout << "The string in array z is: " << z << endl;
27    return 0; // indicates successful termination
28 } // end main
```

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

图 8.31　strcpy 和 strncpy

用 strcat 和 strncat 连接字符串

函数 strcat 把它的第二个参数（一个字符串）追加到它的第一个参数（包含一个字符串的字符数组）中。第二个参数的第一个字符替换了终止第一个参数字符串的空字符（'\0'）。程序员必须保

证用于存储第一个字符串的数组足够大，以便存储第一个字符串、第二个字符串和终止空字符（从第二个字符串复制过来）的联合体。函数strncat把第二个字符串中指定数目的字符追加到第一个字符串，并添加一个终止空字符到结果中。图8.32的程序演示了函数strcat（第19和29行）和函数strncat（第24行）的使用情况。

```
1  // Fig. 8.32: fig08_32.cpp
2  // Using strcat and strncat.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // prototypes for strcat and strncat
8  using std::strcat;
9  using std::strncat;
10
11 int main()
12 {
13    char s1[ 20 ] = "Happy "; // length 6
14    char s2[] = "New Year "; // length 9
15    char s3[ 40 ] = "";
16
17    cout << "s1 = " << s1 << "\ns2 = " << s2;
18
19    strcat( s1, s2 ); // concatenate s2 to s1 (length 15)
20
21    cout << "\n\nAfter strcat(s1, s2):\ns1 = " << s1 << "\ns2 = " << s2;
22
23    // concatenate first 6 characters of s1 to s3
24    strncat( s3, s1, 6 ); // places '\0' after last character
25
26    cout << "\n\nAfter strncat(s3, s1, 6):\ns1 = " << s1
27       << "\ns3 = " << s3;
28
29    strcat( s3, s1 ); // concatenate s1 to s3
30    cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
31       << "\ns3 = " << s3 << endl;
32    return 0; // indicates successful termination
33 } // end main
```

```
s1 = Happy
s2 = New Year

After strcat(s1, s2):
s1 = Happy New Year
s2 = New Year

After strncat(s3, s1, 6):
s1 = Happy New Year
s3 = Happy

After strcat(s3, s1):
s1 = Happy New Year
s3 = Happy Happy New Year
```

图8.32 strcat和strncat

用strcmp和strncmp比较字符串

图8.33用strcmp（第21、22和23行）和strncmp（第26、27和28行）比较三个字符串。函数strcmp逐个字符地比较它的第一个字符串参数和第二个字符串参数。如果这两个字符串相等，函数返回0；如果第一个字符串小于第二个字符串，函数返回一个负值；如果第一个字符串大于第二个

字符串，函数返回一个正值。函数 strncmp 和 strcmp 是等价的，只是 strncmp 仅比较到指定数目的字符。如果遇到其中一个字符串参数的终止空字符，那么函数 strncmp 就停止进行字符的比较。程序打印每次函数调用所返回的整数值。



常见的编程错误 8.21

假定函数 strcmp 和 strncmp 在其参数相等时返回 1（一个 true 值）是一个逻辑错误。这两个函数都返回 0（C++ 的 false 值）表示其参数相等。因此，当检测两个字符串是否相等时，应该把 strcmp 或 strncmp 函数的结果与 0 比较，以确定这两个字符串是否相等。

```
1  // Fig. 8.33: fig08_33.cpp
2  // Using strcmp and strncmp.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 #include <cstring> // prototypes for strcmp and strncmp
11 using std::strcmp;
12 using std::strncmp;
13
14 int main()
15 {
16    char *s1 = "Happy New Year";
17    char *s2 = "Happy New Year";
18    char *s3 = "Happy Holidays";
19
20    cout << "s1 = " << s1 << "\ns2 = " << s2 << "\ns3 = " << s3
21       << "\n\nstrcmp(s1, s2) = " << setw( 2 ) << strcmp( s1, s2 )
22       << "\nstrcmp(s1, s3) = " << setw( 2 ) << strcmp( s1, s3 )
23       << "\nstrcmp(s3, s1) = " << setw( 2 ) << strcmp( s3, s1 );
24
25    cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 )
26       << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = " << setw( 2 )
27       << strncmp( s1, s3, 7 ) << "\nstrncmp(s3, s1, 7) = " << setw( 2 )
28       << strncmp( s3, s1, 7 ) << endl;
29    return 0; // indicates successful termination
30 } // end main
```

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  1
strncmp(s3, s1, 7) = -1
```

图 8.33　strcmp 和 strncmp

为了理解一个字符串“大于”或者“小于”另一个字符串的含义，考虑按字母顺序排列一系列姓氏的过程。毫无疑问，读者会把“Jones”放在“Smith”之前，因为在字母表中“Jones”的第一个字母在“Smith”的第一个字母之前。但是字母表不仅仅只是 26 个字母的清单，它是这些字符的

后输出 sentence 时，注意，只打印了单词“This”，因为在记号化的过程中，strtok 用空字符（'\0'）替换了 sentence 中的每一个空格。

```
// Fig. 8.34: fig08_34.cpp
// Using strtok.
#include <iostream>
using std::cout;
using std::endl;

#include <cstring> // prototype for strtok
using std::strtok;

int main()
{
   char sentence[] = "This is a sentence with 7 tokens";
   char *tokenPtr;

   cout << "The string to be tokenized is:\n" << sentence
      << "\n\nThe tokens are:\n\n";

   // begin tokenization of sentence
   tokenPtr = strtok( sentence, " " );

   // continue tokenizing sentence until tokenPtr becomes NULL
   while ( tokenPtr != NULL )
   {
      cout << tokenPtr << '\n';
      tokenPtr = strtok( NULL, " " ); // get next token
   } // end while

   cout << "\nAfter strtok, sentence = " << sentence << endl;
   return 0; // indicates successful termination
} // end main
```

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:

This
is
a
sentence
with
7
tokens

After strtok, sentence = This
```

图 8.34　strtok



常见的编程错误 8.22

没有意识到 strtok 修改了正在被记号化的字符串，然后试图使用该字符串，就像它是未被修改过的原始字符串一样，这是一个逻辑错误。

确定字符串长度

函数 strlen 以一个字符串作为参数，并返回在该字符串中的字符数——字符串长度不包括终止空字符。该长度值也就是终止空字符在字符串中的索引值。图 8.35 的程序示范了函数 strlen 的用法。

```
1  // Fig. 8.35: fig08_35.cpp
2  // Using strlen.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // prototype for strlen
8  using std::strlen;
9
10 int main()
11 {
12    char *string1 = "abcdefghijklmnopqrstuvwxyz";
13    char *string2 = "four";
14    char *string3 = "Boston";
15
16    cout << "The length of \"" << string1 << "\" is " << strlen( string1 )
17       << "\nThe length of \"" << string2 << "\" is " << strlen( string2 )
18       << "\nThe length of \"" << string3 << "\" is " << strlen( string3 )
19       << endl;
20    return 0; // indicates successful termination
21 } // end main
```

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

图 8.35　strlen 返回一个 char* 字符串的长度

8.14　本章小结

在这一章，我们详细地介绍了指针或者说包含内存地址作为其值的变量。我们从如何声明和初始化指针开始讲起。解释了如何使用地址运算符（&）把一个变量的地址赋给一个指针，如何使用间接运算符（*）访问存储在指针间接引用的变量中的数据。讨论了使用指针参数和引用参数的按引用传递。

我们学习了如何用const修饰指针，以实施最小特权原则。示范了指向非常量数据的非常量指针、指向常量数据的非常量指针、指向非常量数据的常量指针和指向常量数据的常量指针的使用。然后，我们用选择排序来示范按引用传递数组和单个数组元素。讨论了sizeof运算符，可以用它在程序编译期间确定一个数据类型占用的字节大小。

接着我们说明了如何在算术和比较表达式中使用指针。可以看到，可以用指针算术运算，从数组的一个元素跳跃到另外一个元素。我们学习了如何使用指针数组，更明确地说是字符串数组。然后，我们继续讨论了函数指针，它使程序员可以把函数作为参数来传递。在本章末尾，我们讨论了一些操作基于指针字符串的C++函数。学习了一些字符串处理能力，如复制字符串、记号化字符串和确定字符串的长度等。

在下一章，我们开始深入讨论类。将了解类成员的作用域，学习如何使对象处于可靠状态。也将学习使用称为构造函数和析构函数的特殊成员函数，它们分别在一个对象被创建和被销毁时执行。

摘要

- 指针是包含其他变量的内存地址作为其值的变量。
- 声明：

```
int *ptr;
```

声明 ptr 为指向一个 int 类型变量的指针，读做“ptr 是一个指向 int 的指针”。在此声明中所用的 * 指出该变量是一个指针。

在switch语句中，对各种SML指令的行为进行了模拟，如图8.42所示（其他的留给读者）。halt指令还使Simpletron打印每个寄存器的名字和内容，以及打印内存的全部内容。这种输出通常称为计算机转储（computer dump，计算机转储并不是指旧计算机被放置的地方）。为了帮助你编写自己的转储函数，图 8.43 给出了一个说明转储格式的例子。注意，执行 Simpletron 程序之后，计算机转储将显示执行终止时的指令实际值和数据值。为了使数字和其符号以转储中的形式显示，使用流操纵元 showpos。要禁止符号的显示，使用流操纵元noshowpos。对于位数少于4位的数，可以在输出值之前，用以下语句在符号和数值之间加上引导的 0：

```
cout << setfill( '0' ) << internal;
```

当一个不足 4 位的数字以 5 个字符的域宽显示时，参数化的流操纵元 setfill（来自头文件 <iomanip>）指定符号和数值之间的填充字符。（其中一个字符宽度是留给符号使用的）流操纵元 internal 指示填充字符应该出现在符号和数字值之间。

```
读入：    cin >> memory[ operand ];
载入：    accumulator = memory[ operand ];
相加：    accumulator += memory[ operand ];
分支：    稍后我们将讨论分支指令。
停止：    这条指令打印如下的消息：
          *** Simpletron execution terminated ***
```

图 8.42　SML 指令的行为

```
REGISTERS:
accumulator          +0000
counter                00
instructionRegister  +0000
operationCode          00
operand                00

MEMORY:
      0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

图 8.43　一个转储的例子

我们执行程序的第一条指令，即位置 00 中的 +1009。如前所述，switch 语句通过执行以下 C++ 语句模拟这个过程：

```
cin >> memory[ operand ];
```

在执行 cin 语句之前，应该在屏幕上显示一个问号（?），以提示用户输入。Simpletron 等待用户输入一个值并按回车键。然后这个值被读入到内存位置 09 处。

此时，第一条指令的模拟已经完成了。剩下的事就是准备让Simpletron执行下一条指令。由于刚才执行的指令不是一个控制转移，因此我们只需要对指令计数寄存器加 1，如下所示：

```
++counter;
```

这样就完成了第一条指令的模拟执行。接下来整个过程（即指令的执行周期）重新开始，读取下一条要执行的指令。

现在，我们考虑一下如何模拟分支指令（即控制转移指令）。我们所需要做的就是要恰当地调整指令计数器的值。因此，无条件转移指令（40）在 switch 中模拟如下：

```
counter = operand;
```

“如果累加器为 0，则转移”的条件指令可以模拟如下：

```
if ( accumulator == 0 )
   counter = operand;
```

此时，你可以实现你的 Simpletron 模拟程序，并运行在习题 8.18 中写的每个 SML 程序。也许，你还可以增加其他特性来润色 SML，并在你的模拟程序中提供这些特性。

你的模拟程序应该检查各种类型的错误。例如，在装入程序阶段，用户输入到 memory 中的每个数都应当在范围 -9999 到 +9999 之间。你的模拟程序应该用一个 while 循环检测每个输入的数字在这个范围；否则，提示用户重新输入数字，直到用户输入一个正确的数为止。在执行期间，你的模拟程序应该检查各种严重的错误情况。例如，除数为 0 的情况、执行不合法操作码的情况、累加器溢出（即算术运算的结果大于 +9999 或小于 -9999）的情况，等等。这些严重的错误称为致命的错误。当检测到一个致命的错误时，你的模拟程序应该打印以下错误信息：

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

并按照前面讨论的格式打印完整的计算机转储。这可以帮助用户找出程序中的错误。

更多的指针习题

8.20 修改图 8.25 至图 8.27 的洗牌和发牌程序，使洗牌和发牌操作由同一个函数（shuffleAndDeal）完成。该函数应包含类似于图 8.26 中 shuffle 函数的嵌套循环语句。

8.21 请问下面的程序做了什么？

```
1  // Ex. 8.21: ex08_21.cpp
2  // What does this program do?
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  void mystery1( char *, const char * ); // prototype
9
10 int main()
11 {
12    char string1[ 80 ];
13    char string2[ 80 ];
14
15    cout << "Enter two strings: ";
16    cin >> string1 >> string2;
17    mystery1( string1, string2 );
18    cout << string1 << endl;
19    return 0; // indicates successful termination
20 } // end main
21
22 // What does this function do?
```

```
23  void mystery1( char *s1, const char *s2 )
24  {
25     while ( *s1 != '\0' )
26        ++s1;
27
28     for ( ; *s1 = *s2; s1++, s2++ )
29        ; // empty statement
30  } // end function mystery1
```

8.22 请问下面的程序做了什么？

```
 1  // Ex. 8.22: ex08_22.cpp
 2  // What does this program do?
 3  #include <iostream>
 4  using std::cout;
 5  using std::cin;
 6  using std::endl;
 7
 8  int mystery2( const char * ); // prototype
 9
10  int main()
11  {
12     char string1[ 80 ];
13
14     cout << "Enter a string: ";
15     cin >> string1;
16     cout << mystery2( string1 ) << endl;
17     return 0; // indicates successful termination
18  } // end main
19
20  // What does this function do?
21  int mystery2( const char *s )
22  {
23     int x;
24
25     for ( x = 0; *s != '\0'; s++ )
26        ++x;
27
28     return x;
29  } // end function mystery2
```

8.23 指出下列程序段中的错误，并说明如何改正：

a)
```
int *number;
cout << number << endl;
```

b)
```
double *realPtr;
long *integerPtr;
integerPtr = realPtr;
```

c)
```
int * x, y;
x = y;
```

d)
```
char s[] = "this is a character array";
for ( ; *s != '\0'; s++)
   cout << *s << ' ';
```

e)
```
short *numPtr, result;
void *genericPtr = numPtr;
result = *genericPtr + 7 ;
```

f)
```
double x = 19.34;
```

8.25（走迷宫问题）图8.44中由井号(#)和圆点符号(·)组成的网格是一个表示迷宫的二维数组。在这个二维数组中，井号表示迷宫的墙，圆点符号表示可以走出迷宫的可能路径上的一个个方格。只能在数组中包含圆点的地方移动。

对于走迷宫来说，是有一个简单的算法可以保证找到迷宫的出口（假设存在出口）。如果没有出口，则再回到出发点。将你的右手放到你右边的墙上，然后开始向前走。记着手始终不要离开墙。如果迷宫向右转，你也沿着墙向右走。只要你的手不离开墙，最终将到达迷宫的出口。可能还有更短的路径，但是按照这个算法可以保证能走出迷宫。

编写递归函数 mazeTraverse 走迷宫。该函数应该以一个表示迷宫的 12 × 12 的二维数组和迷宫的出发点位置作为参数。mazeTraverse 在试图寻找迷宫的出口时，应该将字符 X 放在沿途的每一格中。函数应该在每次移动之后都显示迷宫，这样用户可以看到迷宫的走法。

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

图 8.44　表示一个迷宫的二维数组

8.26（随机生成迷宫）编写一个函数 mazeGenerator，该函数带一个 12 × 12 的二维字符数组作为参数，并随机产生一个迷宫。同时，该函数应该提供迷宫的起始点和终点的位置。使用几个随机生成的迷宫，测试习题 8.25 的函数 mazeTraverse。

8.27（任意大小的迷宫）将习题8.25和习题8.26的函数mazeTraverse和mazeGenerator一般化，使它们可以处理任意宽和长的迷宫。

8.28（Simpletron 模拟程序的改进）在习题8.19中，你编写了计算机的软件模拟，可以执行用 Simpletron 机器语言（SML）编写的程序。在本题中，我们打算对 Simpletron 模拟程序做一些修改和加强。其中的一些修改和加强，对于本书的姐妹篇《C++大学教程》（第五版）第 21 章习题所构建的编译器产生的程序的执行而言，是必需的。[注意：一些修改可能会相互冲突，因此必须分开操作。]

a) 将 Simpletron 模拟器的内存扩展为包含 1000 个内存位置，使它可以处理更大的程序。
b) 允许 Simpletron 模拟器执行求模计算。这需要增加一条 Simpletron 机器语言指令。
c) 允许 Simpletron 模拟器执行求幂计算，这需要增加一条 Simpletron 机器语言指令。
d) 修改模拟器，使其用十六进制值表示Simpletron机器语言指令，而不是用十进制整数值。
e) 修改模拟器，使其允许输出换行符。这需要增加一条 Simpletron 机器语言指令。
f) 修改模拟器，使其不仅可以处理整数值，还可以处理浮点值。
g) 修改模拟器，使其可以处理字符串输入。[提示：每个Simpletron字可以分为两半，每半都包含一个两位整数。每个两位整数表示一个字符相对应的 ASCII 十进制数值。增加一条 Simpletron 机器语言指令，它可以输入一个字符串并把它存储在以指定的

Simpletron 内存位置开始的内存单元中。该指定位置的前半个字是此字符串中的字符个数（即该字符串的长度）。后续的每半个字都包含了一个用两个十进制数字表示的 ASCII 字符。这条机器语言指令把每个字符转换成它相对应的 ASCII 值，并赋给半个字。]

h) 修改模拟器，使其可以处理按 g) 中格式存储的字符串的输出。[提示：增加一条 Simpletron 机器语言指令，它打印从某个 Simpletron 内存位置开始的一个字符串。该位置的前半个字是字符串中的字符个数（即字符串的长度）。之后的每半个字都包含了一个用两个十进制数字表示的 ASCII 字符。这条机器语言指令检查字符串长度，并通过把每个两位数转换成它们相对应的字符来打印字符串。]

i) 修改模拟器，使其包括指令 SML_DEBUG，它在每一条指令执行完之后打印一个内存转储。SML_DEBUG 的操作码是 44。字 +4401 打开调试模式，而字 +4400 则关闭调试模式。

8.29 指出以下程序的功能。

```
// Ex. 8.29: ex08_29.cpp
// What does this program do?
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

bool mystery3( const char *, const char * ); // prototype

int main()
{
   char string1[ 80 ], string2[ 80 ];

   cout << "Enter two strings: ";
   cin >> string1 >> string2;
   cout << "The result is " << mystery3( string1, string2 ) << endl;
   return 0; // indicates successful termination
} // end main

// What does this function do?
bool mystery3( const char *s1, const char *s2 )
{
   for ( ; *s1 != '\0' && *s2 != '\0'; s1++, s2++ )

      if ( *s1 != *s2 )
         return false;

   return true;
} // end function mystery3
```

字符串操作习题

注意：应该用 C 风格的基于指针的字符串完成以下习题。

8.30 编写一个程序，利用函数 strcmp 比较用户输入的两个字符串。该程序应该指出第一个字符串是否小于、等于或大于第二个字符串。

8.31 编写一个程序，利用函数 strncmp 比较用户输入的两个字符串。该程序应该输入要比较的字符个数，还应该指出第一个字符串是否小于、等于或大于第二个字符串。

8.32 编写一个程序，利用随机数生成创建语句。该程序应该使用 4 个 char 类型的指针数组，它们分别是 article、noun、verb 和 preposition。该程序应当按照如下顺序从每个数组中分别

```
1  // Fig. 9.1: Time.h
2  // Declaration of class Time.
3  // Member functions are defined in Time.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time class definition
10 class Time
11 {
12 public:
13    Time(); // constructor
14    void setTime( int, int, int ); // set hour, minute and second
15    void printUniversal(); // print time in universal-time format
16    void printStandard(); // print time in standard-time format
17 private:
18    int hour; // 0 - 23 (24-hour clock format)
19    int minute; // 0 - 59
20    int second; // 0 - 59
21 }; // end class Time
22
23 #endif
```

图9.1　Time类定义



良好的编程习惯9.1

为了保证程序的清晰性和可读性，每个成员访问说明符在类定义中只使用一次。请将public成员放在最前面，这样更便于查找。



软件工程知识9.1

类的每个元素都应该具有private可见性，除非可以证明它需要public可见性。这是最小权限原则的另一个例子。

请注意在图9.1中，类定义被放入如第5~7行以及第23行所示的如下预处理器封套（preprocessor wrapper）中：

```
// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_H
   ...
#endif
```

当构建比较大的程序时，头文件中还会放入其他的定义和声明。前述的预处理器封套在名字TIME_H已被定义时，可以阻止#ifndef（意思是“如果没有定义”）和#endif之间的代码被包含到文件中。如果以前没有在文件中包含此头文件，那么TIME_H这个名字将被#define指令定义，并且该头文件的语句被包含；如果以前已包含此头文件，那么TIME_H已经定义，该头文件将不再被包含。企图多次包含一个头文件的这种错误，通常（不经意间）发生在具有多个头文件（这些头文件本身可能包含其他的头文件）的大型程序中。注意：预处理器指令中符号常量命名的通常约定是简单地将头文件名用大写形式，其中的圆点用下划线代替。



错误预防技巧9.1

利用预处理器指令#ifndef、#define和#endif等构成预处理器封套，从而避免头文件在一个程序中被多次包含。



良好的编程习惯 9.2
在头文件的预处理器指令 #ifndef 和 #define 中，使用大写头文件名，并用下划线代替圆点。

Time 类的成员函数

在图 9.2 中，Time 构造函数（第 14~17 行）将数据成员初始化为 0（也就是通用时间 12 AM），这就确保了对象可以以一个可靠的状态开始。无效的值不能存放在 Time 对象的数据成员中，因为这个构造函数在 Time 对象创建时被调用，并且客户对数据成员的所有后期改动都是由马上讨论的函数 setTime 完成的。最后，非常重要的一点就是程序员可以为一个类定义多个重载的构造函数。

```
1  // Fig. 9.2: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setfill;
8  using std::setw;
9
10 #include "Time.h" // include definition of class Time from Time.h
11
12 // Time constructor initializes each data member to zero.
13 // Ensures all Time objects start in a consistent state.
14 Time::Time()
15 {
16    hour = minute = second = 0;
17 } // end Time constructor
18
19 // set new Time value using universal time; ensure that
20 // the data remains consistent by setting invalid values to zero
21 void Time::setTime( int h, int m, int s )
22 {
23    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
24    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
25    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
26 } // end function setTime
27
28 // print Time in universal-time format (HH:MM:SS)
29 void Time::printUniversal()
30 {
31    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
32       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
33 } // end function printUniversal
34
35 // print Time in standard-time format (HH:MM:SS AM or PM)
36 void Time::printStandard()
37 {
38    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
39       << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
40       << second << ( hour < 12 ? " AM" : " PM" );
41 } // end function printStandard
```

图 9.2 Time 类成员函数的定义

类的数据成员不能在类体中声明时初始化。我们强烈推荐由类的构造函数初始化这些数据成员（因为对于基本类型的数据成员而言，没有默认的初始化方法）。数据成员也可以用 Time 类的设置函数来赋值。注意：第 10 章展示了仅仅整型类或枚举类型类的 static const 数据成员可以在类的体内初始化。

```
1  // Fig. 9.3: fig09_03.cpp
2  // Program to test class Time.
3  // NOTE: This file must be compiled with Time.cpp.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Time.h" // include definition of class Time from Time.h
9
10 int main()
11 {
12    Time t; // instantiate object t of class Time
13
14    // output Time object t's initial values
15    cout << "The initial universal time is ";
16    t.printUniversal(); // 00:00:00
17    cout << "\nThe initial standard time is ";
18    t.printStandard(); // 12:00:00 AM
19
20    t.setTime( 13, 27, 6 ); // change time
21
22    // output Time object t's new values
23    cout << "\n\nUniversal time after setTime is ";
24    t.printUniversal(); // 13:27:06
25    cout << "\nStandard time after setTime is ";
26    t.printStandard(); // 1:27:06 PM
27
28    t.setTime( 99, 99, 99 ); // attempt invalid settings
29
30    // output t's values after specifying invalid values
31    cout << "\n\nAfter attempting invalid settings:"
32       << "\nUniversal time: ";
33    t.printUniversal(); // 00:00:00
34    cout << "\nStandard time: ";
35    t.printStandard(); // 12:00:00 AM
36    cout << endl;
37    return 0;
38 } // end main
```

```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

图9.3　测试Time类的程序

对象大小

对于那些刚刚接触面向对象编程的人员，他们往往猜想对象一定会很大，因为对象包含了数据成员和成员函数。从逻辑上讲的确如此，程序员可以认为对象是包含了数据和函数（并且我们的讨论自然鼓励这种观点）。可是，事实却不是这样。



性能提示 9.2

对象只包含数据，所以同假设也包含了成员函数的对象相比要小得多。如果对一个类的类名或者该类的一个对象进行sizeof运算的话，结果将只报告该类的数据成员的大小。编译器只创建独立于类的所

有对象的一份成员函数的副本。该类的所有对象共享这份副本。当然，每个对象都需要自己的类数据的副本，因为在对象间这些数据是不同的。函数代码是不可修改的，也称为可重入代码（reentrant code），或者纯过程（pure procedure），因此可以被类的不同对象所共享。

9.3 类的作用域和类成员的访问

类的数据成员（在类定义中声明的变量）和成员函数（在类定义中声明的函数）属于该类的作用域。非成员函数在文件作用域中定义。

在类的作用域内，类的成员可以被类的所有成员函数直接访问，也可以用名字引用。在类的作用域之外，public类成员通过对象的句柄（handles）之一而引用。句柄可以是对象名称、对象的引用或者对象的指针。对象、引用或指针的类型指定了客户可访问的接口（即成员函数）。注意：在第10章我们将看到，每次引用对象中的数据成员或者成员函数时，编译器会插入一个隐式句柄。

类的成员函数可以重载，但是只能由该类的其他成员函数重载。要重载一个成员函数，只需在类定义中提供该重载函数每个版本的原型，同时对每个重载函数的版本提供各自独立的函数定义。

在成员函数中定义的变量具有函数作用域，只有该函数知道它们。如果成员函数定义了与类作用域内变量同名的另一个变量，那么在函数作用域内，函数作用域中的变量将隐藏类作用域中的变量。这样所隐藏的变量可以通过在其名前加类名和二元作用域分辨运算符（::）的方法而访问。同样，隐藏的全局变量可以用一元作用域分辨运算符来访问（参见第6章）。

圆点成员选择运算符（.）前面加对象名称或者对象的引用，则可以访问对象的成员；箭头成员选择运算符（->）前面加对象的指针，则可以访问对象的成员。

图9.4利用一个简单的名为Count的类（第8~25行），举例说明了用上述两种成员选择运算符访问类的成员的方法。此Count类具有一个int类型的private数据成员x（第24行），一个public成员函数setX（第12~15行）以及一个public成员函数print（第18~21行）。出于简单的目的，我们将这个小巧的类以及使用它的main函数放在了同一个文件中。第29~31行生成了3个Count类型的相关变量——counter（一个Count对象）、counterPtr（一个Count对象的指针）和counterRef（一个Count对象的引用）。变量counterRef是对counter的引用，变量counterPtr指向counter。请注意，在第34~35行和第38~39行中，程序使用圆点成员选择运算符（.）前面加对象名称（counter）或对象的引用（counterRef，它是counter的一个别名）调用成员函数setX和print。类似地，第42~43行演示了程序可以通过使用指针（countPtr）和箭头成员选择运算符（->）来调用成员函数setX和print。

```
1  // Fig. 9.4: fig09_04.cpp
2  // Demonstrating the class member access operators . and ->
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // class Count definition
8  class Count
9  {
10 public: // public data is dangerous
11    // sets the value of private data member x
```

图9.4 通过各种对象句柄——对象名称、对象的引用和对象的指针来访问对象的成员函数

```
12    void setX( int value )
13    {
14       x = value;
15    } // end function setX
16
17    // prints the value of private data member x
18    void print()
19    {
20       cout << x << endl;
21    } // end function print
22
23 private:
24    int x;
25 }; // end class Count
26
27 int main()
28 {
29    Count counter; // create counter object
30    Count *counterPtr = &counter; // create pointer to counter
31    Count &counterRef = counter; // create reference to counter
32
33    cout << "Set x to 1 and print using the object's name: ";
34    counter.setX( 1 ); // set data member x to 1
35    counter.print(); // call member function print
36
37    cout << "Set x to 2 and print using a reference to an object: ";
38    counterRef.setX( 2 ); // set data member x to 2
39    counterRef.print(); // call member function print
40
41    cout << "Set x to 3 and print using a pointer to an object: ";
42    counterPtr->setX( 3 ); // set data member x to 3
43    counterPtr->print(); // call member function print
44    return 0;
45 } // end main
```

```
Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3
```

图9.4　通过各种对象句柄——对象名称、对象的引用和对象的指针来访问对象的成员函数（续）

9.4　接口与实现的分离

在第3章中，我们首先是将类的定义和成员函数的定义包含在同一个文件中，然后演示了将这些代码拆分到两个文件中——一个是包含类定义（也就是类的接口）的头文件，另一个是包含类成员函数定义（也就是类的实现）的源代码文件。这样做可以使程序更易于修改——对类的客户来说，只要提供给客户的原始接口保持不变，那么类实现方法的改变就不会对客户造成影响。



软件工程知识 9.6

类的客户要使用类时，不需要访问类的源代码。然而，客户一定需要连接类的目标代码（即类编译后的版本）。这促使独立软件供应商（ISV，Independent Software Vendor）以销售或发放许可证的方式提供各种类库。独立软件供应商在他们的产品中只提供头文件和目标模块，不会透露任何非公开的信息——如果源代码提供的话，则可能会发生。C++用户可以从独立软件供应商提供的越来越多的类库中获得好处。

实际上，事情并不是像这样理想化。头文件的确包含了部分实现细节和有关其他内容的线索。例如，内联成员函数需要放在头文件中，这样一来，当编译器编译客户代码时，该客户代码可以

在适当的位置包含内联函数的定义。头文件的类定义中列出了类的private成员，因此即使客户也许不访问这些 private 成员，它们对于客户而言也是可见的。在第 10 章中，我们将介绍如何使用“代理类”来向类的客户隐藏甚至是类的 private 数据。



软件工程知识 9.7

对于类的接口而言，非常重要的信息应该包含在头文件中；而对于那些只在类内部使用且客户不需要的信息，应该包含在不发布的源文件里。这是最小权限原则的另一个例子。

9.5 访问函数和工具函数

访问函数可以读取或者显示数据。访问函数另一个常见用法是测试条件是真还是假——常常称这样的函数为判定函数。例如，任何容器类都有的isEmpty函数就是判定函数。容器类，例如链表、堆栈或队列等能够容纳许多对象。程序在试图从容器对象中读取一个项目前，可能先要测试isEmpty。isFull判定函数可以测试一个容器类对象，确定它是否还有多余的空间。对于我们的Time类，isAM 和 isPM 就是一组有用的判定函数。

图9.5至图9.7中的程序演示了工具函数的概念（也称为助手函数）。工具函数不属于类的public接口部分。更确切地说，它是支持类的public成员函数操作的private成员函数。工具函数也不是供类的客户使用的（但是正如第 10 章所介绍的，可以被类的友元使用）。

图 9.5 中的 SalesPerson 类声明了代表 12 个月销售额的数组（第 16 行），以及该类的构造函数和操纵该数组的成员函数的原型。

```
1  // Fig. 9.5: SalesPerson.h
2  // SalesPerson class definition.
3  // Member functions defined in SalesPerson.cpp.
4  #ifndef SALESP_H
5  #define SALESP_H
6
7  class SalesPerson
8  {
9  public:
10    SalesPerson(); // constructor
11    void getSalesFromUser(); // input sales from keyboard
12    void setSales( int, double ); // set sales for a specific month
13    void printAnnualSales(); // summarize and print sales
14 private:
15    double totalAnnualSales(); // prototype for utility function
16    double sales[ 12 ]; // 12 monthly sales figures
17 }; // end class SalesPerson
18
19 #endif
```

图 9.5　SalesPerson 类定义

在图 9.6 中，SalesPerson 构造函数（第 15~19 行）将数组 sales 的元素初始化为 0。public 成员函数 setSales（第 36~43 行）设置 sales 数组里某月的销售额。public 成员函数 printAnnualSales（第 46~51 行）打印最近 12 个月的总销售额。private 工具函数 totalAnnualSales（第 54~62 行）为printAnnualSales计算12个月的销售总额。成员函数printAnnualSales将销售额数字编辑为货币格式。

请注意，在图9.7中应用程序的main函数只包含了一个简单的成员函数调用序列——没有任何控制语句。sales 数组的操作逻辑完全封装在 SalesPerson 类的成员函数中。

```
// Fig. 9.6: SalesPerson.cpp
// Member functions for class SalesPerson.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
using std::fixed;

#include <iomanip>
using std::setprecision;

#include "SalesPerson.h" // include SalesPerson class definition

// initialize elements of array sales to 0.0
SalesPerson::SalesPerson()
{
   for ( int i = 0; i < 12; i++ )
      sales[ i ] = 0.0;
} // end SalesPerson constructor

// get 12 sales figures from the user at the keyboard
void SalesPerson::getSalesFromUser()
{
   double salesFigure;

   for ( int i = 1; i <= 12; i++ )
   {
      cout << "Enter sales amount for month " << i << ": ";
      cin >> salesFigure;
      setSales( i, salesFigure );
   } // end for
} // end function getSalesFromUser

// set one of the 12 monthly sales figures; function subtracts
// one from month value for proper subscript in sales array
void SalesPerson::setSales( int month, double amount )
{
   // test for valid month and amount values
   if ( month >= 1 && month <= 12 && amount > 0 )
      sales[ month - 1 ] = amount; // adjust for subscripts 0-11
   else // invalid month or amount value
      cout << "Invalid month or sales figure" << endl;
} // end function setSales

// print total annual sales (with the help of utility function)
void SalesPerson::printAnnualSales()
{
   cout << setprecision( 2 ) << fixed
      << "\nThe total annual sales are: $"
      << totalAnnualSales() << endl; // call utility function
} // end function printAnnualSales

// private utility function to total annual sales
double SalesPerson::totalAnnualSales()
{
   double total = 0.0; // initialize total

   for ( int i = 0; i < 12; i++ ) // summarize sales results
      total += sales[ i ]; // add month i sales to total

   return total;
} // end function totalAnnualSales
```

图 9.6　SalesPerson 类的成员函数定义

```
1  // Fig. 9.7: fig09_07.cpp
2  // Demonstrating a utility function.
3  // Compile this program with SalesPerson.cpp
4
5  // include SalesPerson class definition from SalesPerson.h
6  #include "SalesPerson.h"
7
8  int main()
9  {
10    SalesPerson s; // create SalesPerson object s
11
12    s.getSalesFromUser(); // note simple sequential code;
13    s.printAnnualSales(); // no control statements in main
14    return 0;
15 } // end main
```

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.59
```

图9.7 工具函数演示



软件工程知识9.8

面向对象编程的一个现象是：一旦类被定义，建立并且操作该类对象常常只涉及一个简单的成员函数调用序列——不需要或者只需要少量控制语句。对比之下，在类的成员函数实现中通常含有控制语句。

9.6 Time类实例研究：默认实参的构造函数

图9.8至图9.10中的程序增强了Time类，以演示实参是如何隐式地传递给构造函数的。图9.2中定义的构造函数将hour、minute和second初始化为0（也就是通用时间的午夜12点）。像其他函数一样，构造函数可以指定默认的实参。图9.8中的第13行声明了Time构造函数，包含默认的实参，即为每个传递到构造函数的实参指定默认值为0。在图9.9中，第14~17行定义了Time构造函数的这个新版本。它为形参hr、min和sec接收值，而这些形参分别用于初始化private数据成员hour、minute和second。请注意，这里的Time类为每个数据成员提供了设置和获取函数。Time构造函数现在调用setTime，而后者再调用setHour、setMinute和setSecond函数以确认数据有效性并为数据成员赋值。通过为构造函数提供默认的实参，可以保证即使在构造函数调用时没有提供任何值，构造函数仍然会初始化数据成员，使Time对象处于一个可靠的状态中。一个默认所有实参的构造函数也是一个默认的构造函数，即一个调用时可以不带任何实参的构造函数。每个类最多只有一个默认构造函数。

在图9.9中，第16行构造函数使用传递给它的值（或是默认的值）调用成员函数setTime。而setTime函数首先调用setHour以确保提供给hour的值是在0~23之间，然后调用setMinute和setSecond来确保赋值给minute和second的值是在0~59之间。如果某值超出了范围，这个值将被设置为0（这

样保证每个数据成员都处于可靠的状态）。请注意，通过使用C++的异常处理功能，就有可能“抛出异常”。我们在由电子工业出版社出版的本书的姐妹篇《C++大学教程》（第五版）的第16章中采用了这样的处理。

```
1  // Fig. 9.8: Time.h
2  // Declaration of class Time.
3  // Member functions defined in Time.cpp.
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time abstract data type definition
10 class Time
11 {
12 public:
13    Time( int = 0, int = 0, int = 0 ); // default constructor
14
15    // set functions
16    void setTime( int, int, int ); // set hour, minute, second
17    void setHour( int ); // set hour (after validation)
18    void setMinute( int ); // set minute (after validation)
19    void setSecond( int ); // set second (after validation)
20
21    // get functions
22    int getHour(); // return hour
23    int getMinute(); // return minute
24    int getSecond(); // return second
25
26    void printUniversal(); // output time in universal-time format
27    void printStandard(); // output time in standard-time format
28 private:
29    int hour; // 0 - 23 (24-hour clock format)
30    int minute; // 0 - 59
31    int second; // 0 - 59
32 }; // end class Time
33
34 #endif
```

图9.8　含有默认实参构造函数的Time类

```
1  // Fig. 9.9: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setfill;
8  using std::setw;
9
10 #include "Time.h" // include definition of class Time from Time.h
11
12 // Time constructor initializes each data member to zero;
13 // ensures that Time objects start in a consistent state
14 Time::Time( int hr, int min, int sec )
15 {
16    setTime( hr, min, sec ); // validate and set time
17 } // end Time constructor
18
19 // set new Time value using universal time; ensure that
20 // the data remains consistent by setting invalid values to zero
```

图9.9　含有一个多参构造函数的Time类成员函数定义

```
21 void Time::setTime( int h, int m, int s )
22 {
23    setHour( h ); // set private field hour
24    setMinute( m ); // set private field minute
25    setSecond( s ); // set private field second
26 } // end function setTime
27
28 // set hour value
29 void Time::setHour( int h )
30 {
31    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
32 } // end function setHour
33
34 // set minute value
35 void Time::setMinute( int m )
36 {
37    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
38 } // end function setMinute
39
40 // set second value
41 void Time::setSecond( int s )
42 {
43    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
44 } // end function setSecond
45
46 // return hour value
47 int Time::getHour()
48 {
49    return hour;
50 } // end function getHour
51
52 // return minute value
53 int Time::getMinute()
54 {
55    return minute;
56 } // end function getMinute
57
58 // return second value
59 int Time::getSecond()
60 {
61    return second;
62 } // end function getSecond
63
64 // print Time in universal-time format (HH:MM:SS)
65 void Time::printUniversal()
66 {
67    cout << setfill( '0' ) << setw( 2 ) << getHour() << ":"
68       << setw( 2 ) << getMinute() << ":" << setw( 2 ) << getSecond();
69 } // end function printUniversal
70
71 // print Time in standard-time format (HH:MM:SS AM or PM)
72 void Time::printStandard()
73 {
74    cout << ( ( getHour() == 0 || getHour() == 12 ) ? 12 : getHour() % 12 )
75       << ":" << setfill( '0' ) << setw( 2 ) << getMinute()
76       << ":" << setw( 2 ) << getSecond() << ( hour < 12 ? " AM" : " PM" );
77 } // end function printStandard
```

图 9.9　含有一个多参构造函数的 Time 类成员函数定义（续）

注意，在编写 Time 构造函数时，可以用与成员函数 setTime 相同的语句，或者甚至用 setHour、setMinute 和 setSecond 函数中的各条语句。由构造函数调用 setHour、setMinute 和 setSecond 也许可

以稍稍提高程序的效率，因为省去了对setTime的额外调用。同样，将第31、37、43行代码复制到构造函数中可以省去调用setTime、setHour、setMinute和setSecond的开销。但是，通过复制第31、37、43行代码编写Time构造函数或成员函数setTime会加大维护这个类的难度。如果setHour、setMinute和setSecond的实现方法发生变化，那么任何复制了第31、37、43行代码的成员函数的实现都将需要相应的改变。因此，让Time构造函数调用setTime，setTime调用setHour、setMinute和setSecond，可以使我们将确认hour、minute和second有效性的代码变动局限在相应的设置函数中。这样就可以在改变实现方法时减少出错的可能性。另外，显式声明Time构造函数和setTime为内联函数，或者在类的定义中定义它们（其实这时已经隐式地作为内联函数来定义），也可以提高它们的性能。



软件工程知识 9.9

如果类的成员函数已经提供了类的构造函数（或其他成员函数）所需要的全部或者部分功能，那么就可以在构造函数（或其他成员函数）中调用这样的成员函数。这不仅可以简化代码的维护，而且可以减少由修改代码实现方法所引起出错的可能性。因此，一条普遍原则是：避免重复代码。



软件工程知识 9.10

对函数默认实参值的任何修改都要求重新编译客户代码（以保证程序仍然正常工作）。

图9.10中的main函数初始化了5个Time对象——一个在隐式的构造函数调用时使用了3个默认实参值（第11行）；一个指定了一个实参（第12行）；一个指定了两个实参（第13行）；一个指定了三个实参（第14行）；一个指定了三个无效实参（第15行）。然后，程序分别以通用时间格式和标准时间格式显示每个对象。

```
1  // Fig. 9.10: fig09_10.cpp
2  // Demonstrating a default constructor for class Time.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Time.h" // include definition of class Time from Time.h
8
9  int main()
10 {
11    Time t1; // all arguments defaulted
12    Time t2( 2 ); // hour specified; minute and second defaulted
13    Time t3( 21, 34 ); // hour and minute specified; second defaulted
14    Time t4( 12, 25, 42 ); // hour, minute and second specified
15    Time t5( 27, 74, 99 ); // all bad values specified
16
17    cout << "Constructed with:\n\nt1: all arguments defaulted\n  ";
18    t1.printUniversal(); // 00:00:00
19    cout << "\n  ";
20    t1.printStandard(); // 12:00:00 AM
21
22    cout << "\n\nt2: hour specified; minute and second defaulted\n  ";
23    t2.printUniversal(); // 02:00:00
24    cout << "\n  ";
25    t2.printStandard(); // 2:00:00 AM
26
27    cout << "\n\nt3: hour and minute specified; second defaulted\n  ";
28    t3.printUniversal(); // 21:34:00
29    cout << "\n  ";
30    t3.printStandard(); // 9:34:00 PM
```

图9.10　具有默认实参的构造函数

```
31
32     cout << "\n\nt4: hour, minute and second specified\n  ";
33     t4.printUniversal(); // 12:25:42
34     cout << "\n  ";
35     t4.printStandard(); // 12:25:42 PM
36
37     cout << "\n\nt5: all invalid values specified\n  ";
38     t5.printUniversal(); // 00:00:00
39     cout << "\n  ";
40     t5.printStandard(); // 12:00:00 AM
41     cout << endl;
42     return 0;
43  } // end main
```

```
Constructed with:

t1: all arguments defaulted
  00:00:00
  12:00:00 AM

t2: hour specified; minute and second defaulted
  02:00:00
  2:00:00 AM

t3: hour and minute specified; second defaulted
  21:34:00
  9:34:00 PM

t4: hour, minute and second specified
  12:25:42
  12:25:42 PM

t5: all invalid values specified
  00:00:00
  12:00:00 AM
```

图9.10 具有默认实参的构造函数（续）

关于Time类的设置函数、获取函数和构造函数的补充说明

Time的设置函数和获取函数始终在类内部调用。特别是，setTime函数（见图9.9中的第21~26行）调用了setHour、setMinute和setSecond函数，而printUniversal和printStandard函数在第67~68行和第74~76行分别调用了getHour、getMinute和getSecond函数。在每种情况下，这些函数原本都可以不通过调用这些设置和获取函数而直接访问类的private数据。然而，考虑将时间由现在用3个int值来表达（需要12字节的内存）改变为只用一个int值（表示从当天午夜开始逝去的总秒数，只需要4个字节的内存开销）表达。如果我们进行了这样的改动，那么只有那些直接访问private数据的函数体需要改变——尤其是针对hour、minute和second的各个设置和获取函数。而setTime、printUniversal或者printStandard函数体却不需要修改，因为它们并未直接访问数据。以这样的方式来对类进行设计，可以降低因改变类的实现方法而造成的编程出错的可能性。

类似地，Time类构造函数本来也可以通过复制包含在setTime函数中的适当语句来编写。这样做可能会稍微提高点效率，因为没有了额外的构造函数调用和setTime函数调用的开销。但是，在多个函数或者构造函数中复制语句使得修改类的内部数据表示的难度加大。如果Time构造函数直接调用setTime函数，那么任何对setTime函数实现的改变只需做一次。



常见的编程错误9.2

构造函数可以调用类的其他成员函数，如设置函数或者获取函数等。但是，因为构造函数正在初始化对象，数据成员可能还未处在一个可靠的状态。在数据成员还未适当的初始化之前就使用它们将导致逻辑错误。

它们建立的顺序正好相反。如果用 abort 函数的调用终止程序，那么 static 对象的析构函数将不被调用。

图 9.11 至图 9.13 中的程序演示了不同存储类别的 CreateAndDestroy 类（见图 9.11 和图 9.12）的对象在几种作用域中调用构造函数和析构函数的顺序。每个 CreateAndDestroy 类的对象包含一个整数（objectID）和一个字符串（message）（第 16~17 行），用于程序的输出，以标识该对象。这个机械的例子纯粹是出于教学目的。正因为如此，图 9.12 中析构函数的第 23 行判断是否正被撤销的对象有 objectID 值 1 或者 6，如果是这样，则输出一个换行符。这一行有助于使程序的输出更容易看清楚。

```
1  // Fig. 9.11: CreateAndDestroy.h
2  // Definition of class CreateAndDestroy.
3  // Member functions defined in CreateAndDestroy.cpp.
4  #include <string>
5  using std::string;
6
7  #ifndef CREATE_H
8  #define CREATE_H
9
10 class CreateAndDestroy
11 {
12 public:
13    CreateAndDestroy( int, string ); // constructor
14    ~CreateAndDestroy(); // destructor
15 private:
16    int objectID; // ID number for object
17    string message; // message describing object
18 }; // end class CreateAndDestroy
19
20 #endif
```

图 9.11　CreateAndDestroy 类定义

```
1  // Fig. 9.12: CreateAndDestroy.cpp
2  // Member-function definitions for class CreateAndDestroy.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "CreateAndDestroy.h"// include CreateAndDestroy class definition
8
9  // constructor
10 CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
11 {
12    objectID = ID; // set object's ID number
13    message = messageString; // set object's descriptive message
14
15    cout << "Object " << objectID << "   constructor runs   "
16       << message << endl;
17 } // end CreateAndDestroy constructor
18
19 // destructor
20 CreateAndDestroy::~CreateAndDestroy()
21 {
22    // output newline for certain objects; helps readability
23    cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );
24
25    cout << "Object " << objectID << "   destructor runs    "
26       << message << endl;
27 } // end ~CreateAndDestroy destructor
```

图 9.12　CreateAndDestroy 类的成员函数定义

```
// Fig. 9.13: fig09_13.cpp
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream>
using std::cout;
using std::endl;

#include "CreateAndDestroy.h" // include CreateAndDestroy class definition

void create( void ); // prototype

CreateAndDestroy first( 1, "(global before main)" ); // global object

int main()
{
   cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy second( 2, "(local automatic in main)" );
   static CreateAndDestroy third( 3, "(local static in main)" );

   create(); // call function to create objects

   cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
   CreateAndDestroy fourth( 4, "(local automatic in main)" );
   cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
   return 0;
} // end main

// function to create objects
void create( void )
{
   cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy fifth( 5, "(local automatic in create)" );
   static CreateAndDestroy sixth( 6, "(local static in create)" );
   CreateAndDestroy seventh( 7, "(local automatic in create)" );
   cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
} // end function create
```

```
Object 1   constructor runs   (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2   constructor runs   (local automatic in main)
Object 3   constructor runs   (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5   constructor runs   (local automatic in create)
Object 6   constructor runs   (local static in create)
Object 7   constructor runs   (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7   destructor runs    (local automatic in create)
Object 5   destructor runs    (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4   constructor runs   (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4   destructor runs    (local automatic in main)
Object 2   destructor runs    (local automatic in main)

Object 6   destructor runs    (local static in create)
Object 3   destructor runs    (local static in main)

Object 1   destructor runs    (global before main)
```

图9.13　构造函数和析构函数调用的顺序

图 9.13 在全局作用域中定义了对象 first（第 12 行）。实际上，对于该对象，在 main 中任何语句执行之前，调用它的构造函数；而在运行所有其他对象的析构函数之后程序终止时，调用它的析构函数。

第 14~26 行的 main 函数声明了 3 个对象：对象 second（第 17 行）和 fourth（第 23 行）都是局部自动对象，对象 third（第 18 行）是一个 static 局部对象。当程序执行到每个对象声明处时，调用它们的构造函数。当程序执行到 main 函数的结尾时，依次调用对象 fourth 和 second 的析构函数（即按照它们构造函数调用的相反顺序调用）。因为对象 third 是 static 局部对象，所以直到程序终止时它才撤销。对象 third 的析构函数在全局对象 first 的析构函数调用之前，在所有其他对象撤销之后调用。

第 29~36 行的函数 create 也声明了 3 个对象：fifth（第 32 行）和 seventh（第 34 行）是局部自动对象，sixth（第 33 行）是 static 局部对象。程序执行到 create 函数的结尾时，依次调用对象 seventh 和 fifth 的析构函数（也就是按照其构造函数调用的相反顺序调用）。由于 sixth 是 static 局部对象，因此直到程序终止时它才撤销。sixth 的析构函数的调用发生在 third 和 first 的析构函数调用之前，所有其他对象撤销之后。

9.9 Time 类实例研究：一个微妙的陷阱——返回对 private 数据成员的引用

对象的引用就是该对象名称的别名。因此，它可以在赋值语句的左边使用。在这种情况下，引用成为一个完全令人满意的左值，可以接受一个值。使用这种能力的一条途径（却非常不幸！）是让类的 public 成员函数返回对该类 private 数据成员的引用。请注意，如果函数返回的是一个 const 引用，那么这个引用不能用做可修改的左值。

图 9.14 至图 9.16 中的程序用了一个简化的 Time 类（见图 9.14 和图 9.15）演示如何用成员函数 badSetHour（该函数在图 9.14 中的第 15 行进行了声明，在图 9.15 中的第 29~33 行进行了定义）返回 private 数据成员的引用。实际上，这样的引用返回使成员函数 badSetHour 的调用成为 private 数据成员 hour 的一个别名！此函数调用可以用在 private 数据成员 hour 可用的任何地方，包括作为赋值语句的左值，因而使类的客户可以随心所欲地滥用这个类的 private 数据成员。请注意，当函数返回的是一个指向 private 数据的指针时，同样的问题还会发生。

```
1  // Fig. 9.14: Time.h
2  // Declaration of class Time.
3  // Member functions defined in Time.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  class Time
10 {
11 public:
12    Time( int = 0, int = 0, int = 0 );
13    void setTime( int, int, int );
14    int getHour();
15    int &badSetHour( int ); // DANGEROUS reference return
16 private:
17    int hour;
18    int minute;
19    int second;
20 }; // end class Time
21
22 #endif
```

图 9.14　返回一个对 private 数据成员的引用

```
1  // Fig. 9.15: Time.cpp
2  // Member-function definitions for Time class.
3  #include "Time.h" // include definition of class Time
4
5  // constructor function to initialize private data;
6  // calls member function setTime to set variables;
7  // default values are 0 (see class definition)
8  Time::Time( int hr, int min, int sec )
9  {
10    setTime( hr, min, sec );
11 } // end Time constructor
12
13 // set values of hour, minute and second
14 void Time::setTime( int h, int m, int s )
15 {
16    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
17    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
18    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
19 } // end function setTime
20
21 // return hour value
22 int Time::getHour()
23 {
24    return hour;
25 } // end function getHour
26
27 // POOR PROGRAMMING PRACTICE:
28 // Returning a reference to a private data member.
29 int &Time::badSetHour( int hh )
30 {
31    hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
32    return hour; // DANGEROUS reference return
33 } // end function badSetHour
```

图 9.15　返回一个对 private 数据成员的引用

图 9.16 中的程序首先声明了 Time 对象 t（第 12 行）和引用 hourRef（第 15 行），其中 hourRef 在声明的同时已用调用 t.badSetHour(20)返回的引用进行了初始化。第 17 行显示了别名 hourRef 的值。这恰恰展示了 hourRef 是如何破坏了类的封装性的——main 函数中的语句不应该有访问该类的 private 数据的权利。接下来，第 18 行使用这一别名将 hour 的值设置为 30（一个无效的值），第 19 行显示了由 getHour 函数返回的这个值，由此可见，对 hourRef 的赋值实际上是在修改 Time 对象 t 中的 private 数据。最后，第 23 行使用 badSetHour 函数调用本身作为左值，并将 74（又一个无效值）赋给该函数返回的引用。第 28 行再次显示 getHour 函数返回的值，以说明赋值给第 23 行的函数调用的结果修改了 Time 对象 t 的 private 数据。

```
1  // Fig. 9.16: fig09_16.cpp
2  // Demonstrating a public member function that
3  // returns a reference to a private data member.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Time.h" // include definition of class Time
9
10 int main()
11 {
12    Time t; // create Time object
13
14    // initialize hourRef with the reference returned by badSetHour
```

图 9.16　返回一个对 private 数据成员的引用

```
15     int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour
16
17     cout << "Valid hour before modification: " << hourRef;
18     hourRef = 30; // use hourRef to set invalid value in Time object t
19     cout << "\nInvalid hour after modification: " << t.getHour();
20
21     // Dangerous: Function call that returns
22     // a reference can be used as an lvalue!
23     t.badSetHour( 12 ) = 74; // assign another invalid value to hour
24
25     cout << "\n\n*********************************************************\n"
26        << "POOR PROGRAMMING PRACTICE!!!!!!!!\n"
27        << "t.badSetHour( 12 ) as an lvalue, invalid hour: "
28        << t.getHour()
29        << "\n*********************************************************" << endl;
30     return 0;
31  } // end main
```

```
Valid hour before modification: 20
Invalid hour after modification: 30

*********************************************************
POOR PROGRAMMING PRACTICE!!!!!!!!
t.badSetHour( 12 ) as an lvalue, invalid hour: 74
*********************************************************
```

图 9.16　返回一个对 private 数据成员的引用（续）



错误预防技巧 9.4

返回对private数据成员的引用或指针破坏了类的封装，并使客户代码依赖类的数据的表示。因此，返回对 private 数据的引用或指针是非常危险的行为，应该避免。

9.10　默认的逐个成员赋值

赋值运算符（=）可以将一个对象赋给另一个类型相同的对象。默认情况下，这样的赋值通过逐个成员赋值的方式进行，即赋值运算符右边的对象的每个数据成员逐一赋值给赋值运算符左边对象中的同一数据成员。图 9.17 和图 9.18 定义了这个例子所用的类 Date。图 9.19 中的第 20 行采用默认的逐个成员赋值，将 Date 对象 date1 的数据成员赋给 Date 对象 date2 相应的数据成员。在这种情况下，date1 的 month 成员赋值给 date2 的 month 成员，date1 的 day 成员赋值给 date2 的 day 成员，date1 的 year 成员赋值给 date2 的 year 成员。警告：当所用类的数据成员包含指向动态分配内存的指针时，逐个成员赋值可能会引发严重的问题。我们将在第 11 章中讨论这些问题，并给出解决它们的办法。请注意，这里的 Date 构造函数没有包含任何错误检查措施，我们将此作为课后习题。

```
1  // Fig. 9.17: Date.h
2  // Declaration of class Date.
3  // Member functions are defined in Date.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef DATE_H
7  #define DATE_H
8
9  // class Date definition
10 class Date
11 {
12 public:
13    Date( int = 1, int = 1, int = 2000 ); // default constructor
14    void print();
```

图 9.17　Date 类的头文件

```
15 private:
16    int month;
17    int day;
18    int year;
19 }; // end class Date
20
21 #endif
```

图9.17　Date类的头文件（续）

```
 1 // Fig. 9.18: Date.cpp
 2 // Member-function definitions for class Date.
 3 #include <iostream>
 4 using std::cout;
 5 using std::endl;
 6
 7 #include "Date.h" // include definition of class Date from Date.h
 8
 9 // Date constructor (should do range checking)
10 Date::Date( int m, int d, int y )
11 {
12    month = m;
13    day = d;
14    year = y;
15 } // end constructor Date
16
17 // print Date in the format mm/dd/yyyy
18 void Date::print()
19 {
20    cout << month << '/' << day << '/' << year;
21 } // end function print
```

图9.18　Date类的成员函数定义

```
 1 // Fig. 9.19: fig09_19.cpp
 2 // Demonstrating that class objects can be assigned
 3 // to each other using default memberwise assignment.
 4 #include <iostream>
 5 using std::cout;
 6 using std::endl;
 7
 8 #include "Date.h" // include definition of class Date from Date.h
 9
10 int main()
11 {
12    Date date1( 7, 4, 2004 );
13    Date date2; // date2 defaults to 1/1/2000
14
15    cout << "date1 = ";
16    date1.print();
17    cout << "\ndate2 = ";
18    date2.print();
19
20    date2 = date1; // default memberwise assignment
21
22    cout << "\n\nAfter default memberwise assignment, date2 = ";
23    date2.print();
24    cout << endl;
25    return 0;
26 } // end main
```

```
date1 = 7/4/2004
date2 = 1/1/2000

After default memberwise assignment, date2 = 7/4/2004
```

图9.19　默认的逐个成员赋值



常见的编程错误 10.3

const 对象调用非 const 成员函数将导致编译错误。



软件工程知识 10.2

可以对const成员函数进行非const版本的重载。编译器将根据调用函数的对象性质选择相应的重载函数来使用。如果对象是 const 的，则编译器使用 const 版本的重载函数；如果对象是非 const 的，则编译器使用非 const 版本的重载函数。

对于构造函数和析构函数来说，一个有趣的问题是两者都典型地修改对象。请记住不允许对构造函数和析构函数进行 const 声明。必须允许构造函数修改对象，这样对象才能恰当地初始化。而析构函数必须能够在对象使用的内存被系统回收之前进行它的扫尾工作。



常见的编程错误 10.4

试图声明构造函数和析构函数为 const 是一个编译错误。

定义和使用 const 成员函数

图 10.1 到图 10.3 的程序修改了图 9.9 和图 9.10 的 Time 类，将其中的获取函数和 printUniversal 函数声明为 const。在头文件 Time.h（见图 10.1）中，第 19~21 行和第 24 行现在在每个函数的参数列表后面包含了关键字 const。在图 10.2 中每个函数的相应定义中都在函数的参数列表后指定了关键字 const（第 47、53、59 和 65 行）。

```
1  // Fig. 10.1: Time.h
2  // Definition of class Time.
3  // Member functions defined in Time.cpp.
4  #ifndef TIME_H
5  #define TIME_H
6
7  class Time
8  {
9  public:
10    Time( int = 0, int = 0, int = 0 ); // default constructor
11
12    // set functions
13    void setTime( int, int, int ); // set time
14    void setHour( int ); // set hour
15    void setMinute( int ); // set minute
16    void setSecond( int ); // set second
17
18    // get functions (normally declared const)
19    int getHour() const; // return hour
20    int getMinute() const; // return minute
21    int getSecond() const; // return second
22
23    // print functions (normally declared const)
24    void printUniversal() const; // print universal time
25    void printStandard(); // print standard time (should be const)
26 private:
27    int hour; // 0 - 23 (24-hour clock format)
28    int minute; // 0 - 59
29    int second; // 0 - 59
30 }; // end class Time
31
32 #endif
```

图 10.1　具有 const 成员函数的 Time 类的定义

图 10.3 实例化了两个 Time 对象——非 const 对象 wakeUp（第 7 行）和 const 对象 noon（第 8 行）。程序试图对 const 对象 noon 调用非 const 的成员函数 setHour（第 13 行）和 printStandard（第 20 行）。

两种情况下，编译器都产生了一条错误消息。这个程序也展现了对对象进行的三种其他的成员函数调用——对非 const 对象调用非 const 成员函数（第 11 行），对非 const 对象调用 const 成员函数（第 15 行）和对 const 对象调用 const 成员函数（第 17~18 行）。对 const 对象调用非 const 成员函数而产生的出错消息显示在输出窗口中。请注意，虽然当前一些编译器对于第 13 行和第 20 行只产生警告消息（这样仍然允许程序执行），但是我们认为这些警告是错误的—— ANSI/ISO C++ 标准不允许对 const 对象调用非 const 成员函数。

```
1  // Fig. 10.2: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setfill;
8  using std::setw;
9
10 #include "Time.h" // include definition of class Time
11
12 // constructor function to initialize private data;
13 // calls member function setTime to set variables;
14 // default values are 0 (see class definition)
15 Time::Time( int hour, int minute, int second )
16 {
17    setTime( hour, minute, second );
18 } // end Time constructor
19
20 // set hour, minute and second values
21 void Time::setTime( int hour, int minute, int second )
22 {
23    setHour( hour );
24    setMinute( minute );
25    setSecond( second );
26 } // end function setTime
27
28 // set hour value
29 void Time::setHour( int h )
30 {
31    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
32 } // end function setHour
33
34 // set minute value
35 void Time::setMinute( int m )
36 {
37    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
38 } // end function setMinute
39
40 // set second value
41 void Time::setSecond( int s )
42 {
43    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
44 } // end function setSecond
45
46 // return hour value
47 int Time::getHour() const // get functions should be const
48 {
49    return hour;
50 } // end function getHour
51
52 // return minute value
```

图 10.2　包含了 const 成员函数的 Time 类的成员函数定义

```
53 int Time::getMinute() const
54 {
55    return minute;
56 } // end function getMinute
57
58 // return second value
59 int Time::getSecond() const
60 {
61    return second;
62 } // end function getSecond
63
64 // print Time in universal-time format (HH:MM:SS)
65 void Time::printUniversal() const
66 {
67    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
68       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
69 } // end function printUniversal
70
71 // print Time in standard-time format (HH:MM:SS AM or PM)
72 void Time::printStandard() // note lack of const declaration
73 {
74    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
75       << ":" << setfill( '0' ) << setw( 2 ) << minute
76       << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
77 } // end function printStandard
```

图 10.2　包含了 const 成员函数的 Time 类的成员函数定义（续）

还要注意，尽管构造函数必须是非 const 函数（见图 10.2 中的第 15~18 行），但它仍然可以用来初始化 const 对象（见图 10.3 中的第 8 行）。Time 类构造函数的定义（见图 10.2 中的第 15~18 行）显示 Time 类构造函数调用了另一个非 const 成员函数 setTime（第 21~26 行），以完成 Time 对象的初始化。在构造函数中调用非 const 成员函数来作为初始化 const 对象的一部分是允许的。const 对象的“常量性质”在从构造函数完成对象初始化开始到对象的析构函数被调用结束这个过程之间，一直是保持的。

另外，注意图 10.3 中第 20 行，虽然 Time 类的成员函数 printStandard 并未修改调用它的对象，可是仍产生了一个编译错误。成员函数不修改对象的事实并不足以指明该函数是 const 函数——函数必须显式地声明为 const。

```
1  // Fig. 10.3: fig10_03.cpp
2  // Attempting to access a const object with non-const member functions.
3  #include "Time.h" // include Time class definition
4
5  int main()
6  {
7     Time wakeUp( 6, 45, 0 ); // non-constant object
8     const Time noon( 12, 0, 0 ); // constant object
9
10                              // OBJECT      MEMBER FUNCTION
11    wakeUp.setHour( 18 );     // non-const   non-const
12
13    noon.setHour( 12 );       // const       non-const
14
15    wakeUp.getHour();         // non-const   const
16
17    noon.getMinute();         // const       const
18    noon.printUniversal();    // const       const
19
20    noon.printStandard();     // const       non-const
21    return 0;
```

图 10.3　const 对象和 const 成员函数

```
22  } // end main
```

Borland C++ command-line compiler error messages:

```
Warning W8037 fig10_03.cpp 13: Non-const function Time::setHour(int)
   called for const object in function main()
Warning W8037 fig10_03.cpp 20: Non-const function Time::printStandard()
   called for const object in function main()
```

Microsoft Visual C++.NET compiler error messages:

```
C:\scpphtp5_examples\ch10\Fig10_01_03\fig10_03.cpp(13) : error C2662:
   'Time::setHour' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
       Conversion loses qualifiers
C:\scpphtp5_examples\ch10\Fig10_01_03\fig10_03.cpp(20) : error C2662:
   'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
       Conversion loses qualifiers
```

GNU C++ compiler error messages:

```
fig10_03.cpp:13: error: passing `const Time' as `this' argument of
   `void Time::setHour(int)' discards qualifiers
fig10_03.cpp:20: error: passing `const Time' as `this' argument of
   `void Time::printStandard()' discards qualifiers
```

图 10.3　const 对象和 const 成员函数（续）

用成员初始化器来初始化 const 数据成员

图 10.4 至图 10.6 中的程序介绍使用成员初始化器的语法（member initializer syntax）。所有的数据成员均可以用成员初始化器形式进行初始化，但是 const 数据成员和引用的数据成员必须使用成员初始化器进行初始化。在这一章中，我们将看到成员对象也必须用这种方法进行初始化。在第 12 章，当我们学习继承时，还会看到派生类的基类部分也必须以这种方法进行初始化。

```
1  // Fig. 10.4: Increment.h
2  // Definition of class Increment.
3  #ifndef INCREMENT_H
4  #define INCREMENT_H
5
6  class Increment
7  {
8  public:
9     Increment( int c = 0, int i = 1 ); // default constructor
10
11    // function addIncrement definition
12    void addIncrement()
13    {
14       count += increment;
15    } // end function addIncrement
16
17    void print() const; // prints count and increment
18 private:
19    int count;
20    const int increment; // const data member
21 }; // end class Increment
22
23 #endif
```

图 10.4　包含非 const 数据成员 count 和 const 数据成员 increment 的 Increment 类定义

```
1  // Fig. 10.5: Increment.cpp
2  // Member-function definitions for class Increment demonstrate using a
3  // member initializer to initialize a constant of a built-in data type.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Increment.h" // include definition of class Increment
9
10 // constructor
11 Increment::Increment( int c, int i )
12    : count( c ), // initializer for non-const member
13      increment( i ) // required initializer for const member
14 {
15    // empty body
16 } // end constructor Increment
17
18 // print count and increment values
19 void Increment::print() const
20 {
21    cout << "count = " << count << ", increment = " << increment << endl;
22 } // end function print
```

图 10.5 使用成员初始化器初始化内置数据类型的常量

```
1  // Fig. 10.6: fig10_06.cpp
2  // Program to test class Increment.
3  #include <iostream>
4  using std::cout;
5
6  #include "Increment.h" // include definition of class Increment
7
8  int main()
9  {
10    Increment value( 10, 5 );
11
12    cout << "Before incrementing: ";
13    value.print();
14
15    for ( int j = 1; j <= 3; j++ )
16    {
17       value.addIncrement();
18       cout << "After increment " << j << ": ";
19       value.print();
20    } // end for
21
22    return 0;
23 } // end main
```

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
```

图 10.6 调用 Increment 对象的 print 和 addIncrement 成员函数

构造函数的定义（见图 10.5 中的第 11~16 行）使用一个成员初始化器列表（member initializer list）来初始化 Increment 类的数据成员——非 const 整数 count 和 const 整数 increment（在图 10.4 的第 19~20 行中已声明）。成员初始化器出现在构造函数的参数列表和作为构造函数体开始的左花括号之间。成员初始化器列表（见图 10.5 中的第 12~13 行）用一个冒号（:）与参数列表分开。每个成员初始化器都是由数据成员名称后面紧跟着包含该成员初始化值的一对圆括号组成。在这个例子

中，count用构造函数参数c的值来初始化，increment用构造函数参数i的值进行初始化。请注意多个成员初始化器之间用逗号隔开。另外也请注意成员初始化器列表在构造函数体执行之前被执行。



软件工程知识 10.3

const对象不能通过赋值修改，所以它必须被初始化。当类的某个数据成员声明为const时，就必须用相应的成员初始化器，以便向构造函数提供该类对象的这一数据成员的初始化值。对于引用也是一样的。

错误地企图用赋值初始化 const 数据成员

图 10.7 至图 10.9 中的程序举例说明了试图在 Increment 构造函数体内使用赋值语句（见图 10.8 中的第 14 行）初始化 const 数据成员 increment，而不是用成员初始化器造成的编译错误。请注意，图10.8中的第13行不会产生一个编译错误，因为count没有被声明为const。另外也注意到Microsoft Visual C++.NET 产生的编译错误信息中 int 数据成员 increment 为“const object”（即 const 对象）。ANSI/ISO C++标准将“对象”定义为任何“存储的区域”。就像类的实例一样，基本类型变量同样在内存中占有空间，所以它们常被称为“对象”。



常见的编程错误 10.5

不为 const 数据成员提供相应的成员初始化器是一个编译错误。



软件工程知识 10.4

常量数据成员（const对象和const变量）和声明为引用的数据成员必须采用成员初始化器语法形式进行初始化，在构造函数中为这些类型数据赋值是不允许的。

```
1  // Fig. 10.7: Increment.h
2  // Definition of class Increment.
3  #ifndef INCREMENT_H
4  #define INCREMENT_H
5
6  class Increment
7  {
8  public:
9     Increment( int c = 0, int i = 1 ); // default constructor
10
11    // function addIncrement definition
12    void addIncrement()
13    {
14       count += increment;
15    } // end function addIncrement
16
17    void print() const; // prints count and increment
18 private:
19    int count;
20    const int increment; // const data member
21 }; // end class Increment
22
23 #endif
```

图 10.7　包含非 const 数据成员 count 和 const 数据成员 increment 的 Increment 类定义

请注意这里讲到print函数（见图10.8中的第18~21行）声明为const。也许将这个函数标记为const会使人感到很奇怪，因为程序可能永远不会拥有const的Increment对象。可是，程序却有可能具有Increment对象的const引用或者指向Increment对象的const指针。通常，这种情况发生在将Increment类的对象传递给函数或从函数返回Increment类的对象时。这时，只有Increment类的const成员函数可以通过引用或指针被调用。因此，将print函数声明为const是合理的——这样做可以避免在将 Increment 对象作为 const 对象进行处理的过程中产生的错误。



错误预防技巧 10.1

请将所有运行时不修改对象的成员函数都声明为 const。有时候，这样的做法似乎不合时宜，因为你可能没有创建该类 const 对象的打算，也没有通过 const 引用或 const 指针访问该类对象的意图。不过将这样的成员函数声明为 const 确实有一个好处。如果写成员函数时不经意地修改了对象，那么编译器将产生一条错误消息。

```
1  // Fig. 10.8: Increment.cpp
2  // Attempting to initialize a constant of
3  // a built-in data type with an assignment.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Increment.h" // include definition of class Increment
9
10 // constructor; constant member 'increment' is not initialized
11 Increment::Increment( int c, int i )
12 {
13    count = c; // allowed because count is not constant
14    increment = i; // ERROR: Cannot modify a const object
15 } // end constructor Increment
16
17 // print count and increment values
18 void Increment::print() const
19 {
20    cout << "count = " << count << ", increment = " << increment << endl;
21 } // end function print
```

图 10.8　错误地试图用赋值的方式初始化一个内置数据类型的常量

```
1  // Fig. 10.9: fig10_09.cpp
2  // Program to test class Increment.
3  #include <iostream>
4  using std::cout;
5
6  #include "Increment.h" // include definition of class Increment
7
8  int main()
9  {
10    Increment value( 10, 5 );
11
12    cout << "Before incrementing: ";
13    value.print();
14
15    for ( int j = 1; j <= 3; j++ )
16    {
17       value.addIncrement();
18       cout << "After increment " << j << ": ";
19       value.print();
20    } // end for
21
22    return 0;
23 } // end main
```

Borland C++ command-line compiler error message:

```
Error E2024 Increment.cpp 14: Cannot modify a const object in function
   Increment::Increment(int,int)
```

图 10.9　测试 Increment 类的程序产生编译错误

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.cpp(12) : error C2758:
   'Increment::increment' : must be initialized in constructor
   base/member initializer list
         C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.h(20) :
            see declaration of 'Increment::increment'
C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.cpp(14) : error C2166:
   l-value specifies const object
```

GNU C++ compiler error messages:

```
Increment.cpp:12: error: uninitialized member 'Increment::increment' with
   'const' type 'const int'
Increment.cpp:14: error: assignment of read-only data-member
   `Increment::increment'
```

图 10.9　测试 Increment 类的程序产生编译错误（续）

10.3　组成：对象作为类的成员

AlarmClock 对象需要知道什么时候要闹铃响起，因此为什么不将一个 Time 对象纳入 AlarmClock 类的定义之中，作为它的一个成员呢？这种功能称为组成（composition），有时也称为“有一个”关系（has-a relationship）。一个类可以将其他类的对象作为其成员。



软件工程知识 10.5

软件重用性的一个普遍的形式是组成，即一个类将其他类的对象作为成员。

当创建对象时，构造函数自动被调用。以前，我们看到怎样将参数传递给在 main 函数内所创建对象的构造函数。在这一节我们将介绍构造函数如何通过成员初始化器完成将参数传递给成员对象构造函数的任务。成员对象以在类的定义中声明的顺序（不是以在构造函数的成员初始化器列表中列出的顺序）在包含他们的对象，有时称为宿主对象（host objects））构造之前建立。

图 10.10 至图 10.14 的程序使用 Date 类（见图 10.10 至图 10.11）以及 Employee 类（见图 10.12 至图 10.13）演示作为其他对象成员的对象。Employee 对象的定义（见图 10.12）包含 private 数据成员 firstName、lastName、birthDate 和 hireDate。成员 birthDate 和 hireDate 是 const 的 Date 类对象，包含了 private 数据成员 month、day 和 year。Employee 类构造函数的头部（见图 10.13 中的第 18~21 行）指示该构造函数接收四个参数（first、last、dateOfBirth 和 dateOfHire）。前两个参数被用来在构造函数体中初始化字符数组 firstName 和 lastName。后两个参数通过成员初始化器传递给 Date 类的构造函数。在头部中的冒号（:）将参数列表和成员初始化器隔开。成员初始化器指定了传递给 Date 类型成员对象构造函数的那些 Employee 构造函数参数。参数 dateOfBirth 被传递给 birthDate 对象的构造函数（见图 10.13 中的第 20 行），参数 dateOfHire 被传递给 hireDate 对象的构造函数（见图 10.13 中的第 21 行）。同样，成员初始化器用逗号分隔开来。当你学习 Date 类（见图 10.10）时，请注意这个类并没有提供接收一个 Date 类型参数的构造函数。既然如此，那么 Employee 类构造函数中的成员初始化器列表是怎样通过将 Date 对象的参数传递给 Date 构造函数，来初始化对象 birthDate 和 hireDate？就像我们在第9章里提到的一样，编译器提供给每个类一个默认的拷贝构造函数，该函数将构造函数的参数对象的每个成员复制给将要初始化对象的相应成员。第 11 章将讨论程序员如何定义自己的拷贝构造函数。

图 10.14 创建两个 Date 对象（第 11~12 行），然后将它们作为参数传递给第 13 行中建立的 Employee 对象的构造函数。第16行输出这个 Employee 对象的数据。当第 11~12 行中的每个 Date 对

象被创建时，图10.11中第11~28行定义的Date构造函数显示一行输出信息表示构造函数已经被调用过了（参见范例输出的前两行）。注意：图10.14的第13行造成两次另外的Date构造函数的调用，它们没有在程序的输出中留下痕迹。当Employee的Date成员对象在Employee构造函数的成员初始化器列表中初始化时，Date类默认的拷贝构造函数被调用。该构造函数由编译器隐式地定义，并且不包含任何在被其调用时要显示的输出语句。在第11章中我们将详细讨论构造函数和默认拷贝构造函数。

```
1  // Fig. 10.10: Date.h
2  // Date class definition; Member functions defined in Date.cpp
3  #ifndef DATE_H
4  #define DATE_H
5
6  class Date
7  {
8  public:
9     Date( int = 1, int = 1, int = 1900 ); // default constructor
10    void print() const; // print date in month/day/year format
11    ~Date(); // provided to confirm destruction order
12 private:
13    int month; // 1-12 (January-December)
14    int day; // 1-31 based on month
15    int year; // any year
16
17    // utility function to check if day is proper for month and year
18    int checkDay( int ) const;
19 }; // end class Date
20
21 #endif
```

图10.10 Date类的定义

```
1  // Fig. 10.11: Date.cpp
2  // Member-function definitions for class Date.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Date.h" // include Date class definition
8
9  // constructor confirms proper value for month; calls
10 // utility function checkDay to confirm proper value for day
11 Date::Date( int mn, int dy, int yr )
12 {
13    if ( mn > 0 && mn <= 12 ) // validate the month
14       month = mn;
15    else
16    {
17       month = 1; // invalid month set to 1
18       cout << "Invalid month (" << mn << ") set to 1.\n";
19    } // end else
20
21    year = yr; // could validate yr
22    day = checkDay( dy ); // validate the day
23
24    // output Date object to show when its constructor is called
25    cout << "Date object constructor for date ";
26    print();
27    cout << endl;
28 } // end Date constructor
29
```

图10.11 Date类成员函数的定义

```
30  // print Date object in form month/day/year
31  void Date::print() const
32  {
33     cout << month << '/' << day << '/' << year;
34  } // end function print
35
36  // output Date object to show when its destructor is called
37  Date::~Date()
38  {
39     cout << "Date object destructor for date ";
40     print();
41     cout << endl;
42  } // end ~Date destructor
43
44  // utility function to confirm proper day value based on
45  // month and year; handles leap years, too
46  int Date::checkDay( int testDay ) const
47  {
48     static const int daysPerMonth[ 13 ] =
49        { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
50
51     // determine whether testDay is valid for specified month
52     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
53        return testDay;
54
55     // February 29 check for leap year
56     if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
57        ( year % 4 == 0 && year % 100 != 0 ) ) )
58        return testDay;
59
60     cout << "Invalid day (" << testDay << ") set to 1.\n";
61     return 1; // leave object in consistent state if bad value
62  } // end function checkDay
```

图 10.11　Date类成员函数的定义（续）

```
1  // Fig. 10.12: Employee.h
2  // Employee class definition.
3  // Member functions defined in Employee.cpp.
4  #ifndef EMPLOYEE_H
5  #define EMPLOYEE_H
6
7  #include "Date.h" // include Date class definition
8
9  class Employee
10 {
11 public:
12    Employee( const char * const, const char * const,
13       const Date &, const Date & );
14    void print() const;
15    ~Employee(); // provided to confirm destruction order
16 private:
17    char firstName[ 25 ];
18    char lastName[ 25 ];
19    const Date birthDate; // composition: member object
20    const Date hireDate; // composition: member object
21 }; // end class Employee
22
23 #endif
```

图 10.12　展示组成Employee类的定义

```
1  // Fig. 10.13: Employee.cpp
2  // Member-function definitions for class Employee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // strlen and strncpy prototypes
8  using std::strlen;
9  using std::strncpy;
10
11 #include "Employee.h" // Employee class definition
12 #include "Date.h" // Date class definition
13
14 // constructor uses member initializer list to pass initializer
15 // values to constructors of member objects birthDate and hireDate
16 // [Note: This invokes the so-called "default copy constructor" which the
17 // C++ compiler provides implicitly.]
18 Employee::Employee( const char * const first, const char * const last,
19    const Date &dateOfBirth, const Date &dateOfHire )
20    : birthDate( dateOfBirth ), // initialize birthDate
21      hireDate( dateOfHire ) // initialize hireDate
22 {
23    // copy first into firstName and be sure that it fits
24    int length = strlen( first );
25    length = ( length < 25 ? length : 24 );
26    strncpy( firstName, first, length );
27    firstName[ length ] = '\0';
28
29    // copy last into lastName and be sure that it fits
30    length = strlen( last );
31    length = ( length < 25 ? length : 24 );
32    strncpy( lastName, last, length );
33    lastName[ length ] = '\0';
34
35    // output Employee object to show when constructor is called
36    cout << "Employee object constructor: "
37       << firstName << ' ' << lastName << endl;
38 } // end Employee constructor
39
40 // print Employee object
41 void Employee::print() const
42 {
43    cout << lastName << ", " << firstName << "  Hired: ";
44    hireDate.print();
45    cout << "  Birthday: ";
46    birthDate.print();
47    cout << endl;
48 } // end function print
49
50 // output Employee object to show when its destructor is called
51 Employee::~Employee()
52 {
53    cout << "Employee object destructor: "
54       << lastName << ", " << firstName << endl;
55 } // end ~Employee destructor
```

图 10.13　Employee 类成员函数的定义，包含带有一张成员初始化器列表的构造函数

Date 类和 Employee 类各自都包含一个析构函数（分别见图 10.11 中的第 37~42 行和图 10.13 中的第 51~55 行）。当这两个类的对象被撤销时，析构函数打印一条消息。这使我们可以在程序的输出中确认对象是由内而外进行创建的，而撤销则是按相反的顺序，即由外而内进行的（换言之，Date 成员对象在包含它们的 Employee 对象撤销后再撤销）。请注意图 10.14 输出的最后四行。最后的两行分别是运行于 Date 对象 hire（第 12 行）和 birth（第 11 行）的 Date 析构函数的输出。这些输

出证实了 main 函数内创建的三个对象是以它们构建的相反顺序进行撤销的（Employee 析构函数输出是底下的五行）。输出窗口中倒数第 4、3 行分别显示 Employee 的成员对象 hireDate（见图 10.12 的第 20 行）和 birthDate（见图 10.12 的第 19 行）析构函数的运行。这些输出确认 Employee 对象是由外而内进行撤销的——即，Employee 析构函数首先运行（在输出窗口倒数五行显示），然后成员对象再以与它们构造相反的顺序撤销。同样图 10.14 的输出中未显示这些对象构造函数运行信息，因为这些构造函数是由 C++ 编译器提供的默认拷贝构造函数。

```
1  // Fig. 10.14: fig10_14.cpp
2  // Demonstrating composition--an object with member objects.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Employee.h" // Employee class definition
8
9  int main()
10 {
11    Date birth( 7, 24, 1949 );
12    Date hire( 3, 12, 1988 );
13    Employee manager( "Bob", "Blue", birth, hire );
14
15    cout << endl;
16    manager.print();
17
18    cout << "\nTest Date constructor with invalid values:\n";
19    Date lastDayOff( 14, 35, 1994 ); // invalid month and day
20    cout << endl;
21    return 0;
22 } // end main
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Blue

Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949

Test Date constructor with invalid values:
Invalid month (14) set to 1.
Invalid day (35) set to 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

图 10.14　成员对象的初始化程序

成员对象不需要显式地通过成员初始化器进行初始化。如果没有提供成员初始化器，成员对象的默认构造函数将被隐式调用。由默认构造函数建立的值，如果有的话，可以用 set 函数重载。但是，对于复杂的初始化，这种方法可能需要大量额外的工作和时间。



常见的编程错误 10.6

如果成员对象不是用成员初始化器形式进行初始化并且成员对象的类没有提供默认的构造函数（换言之，成员对象的类定义了一个或多个构造函数，但是没有一个是默认的构造函数），则将产生一个编译错误。

```
friend class ClassTwo;
```



软件工程知识 10.7
即使 friend 函数的原型在类定义内出现，友元仍不是成员函数。



软件工程知识 10.8
private、protected 和 public 这些成员访问说明符标志与友元的声明无关，因此友元定义可以放在类定义内的任何地方。



良好的编程习惯 10.1
在类定义体之内把所有的友元关系声明放在最前面的位置，并且不要在其前面添加任何成员访问说明符。

友元关系是授予的，而不是索取的——也就是说，对于类 B 成为类 A 的友元，类 A 必须显式地声明类 B 是它的友元。另外，友元关系既不是对称的也不是传递的，也就是说，如果类 A 是类 B 的友元，类 B 是类 C 的友元，那么就不能推断类 B 是类 A 的友元（说明了友元关系不是对称的），类 C 是类 B 的友元（同样是因为友元是不对称的），或者类 A 是类 C 的友元（友元关系是不传递的）。



软件工程知识 10.9
OOP 组织中的一些人认为“友元关系”破坏了信息的隐藏并削弱了面向对象设计方法的优势。在本书中，我们将举一些友元关系可靠应用的例子。

用 friend 函数修改类的 private 数据

图 10.15 是一个非常机械的例子，在这个例子中，我们定义 friend 函数 setX 来设置 Count 类的 private 数据成员 x。请注意，友元声明（第 10 行）首先（按照惯例）出现在类的定义中，甚至出现在 public 成员函数声明之前。再次说明，友元声明可以出现在类的任何地方。

```
1  // Fig. 10.15: fig10_15.cpp
2  // Friends can access private members of a class.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // Count class definition
8  class Count
9  {
10    friend void setX( Count &, int ); // friend declaration
11 public:
12    // constructor
13    Count()
14       : x( 0 ) // initialize x to 0
15    {
16       // empty body
17    } // end constructor Count
18
19    // output x
20    void print() const
21    {
22       cout << x << endl;
23    } // end function print
24 private:
25    int x; // data member
26 }; // end class Count
27
```

图 10.15　友元可以访问类的 private 成员

```
28 // function setX can modify private data of Count
29 // because setX is declared as a friend of Count (line 10)
30 void setX( Count &c, int val )
31 {
32    c.x = val; // allowed because setX is a friend of Count
33 } // end function setX
34
35 int main()
36 {
37    Count counter; // create Count object
38
39    cout << "counter.x after instantiation: ";
40    counter.print();
41
42    setX( counter, 8 ); // set x using a friend function
43    cout << "counter.x after call to setX friend function: ";
44    counter.print();
45    return 0;
46 } // end main
```

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

图 10.15　友元可以访问类的 private 成员（续）

函数 setX（第 30~33 行）是一个 C 语言风格的独立函数——不是 Count 类的成员函数。因此，对于 counter 对象，当 setX 被调用时，第 42 行将 counter 作为参数传递给 setX，而不是像如下语句：

```
counter.setX( 8 );
```

一样使用一个句柄（例如该对象的名称）来调用此函数。我们曾经提到，图 10.15 是使用友元概念的一个非常机械的范例。正常情况下，适合将 setX 函数定义为 Count 类的成员函数。而且，也适合将图 10.15 中的程序分割为如下三个文件：

1. 一个头文件（例如，Count.h），包含 Count 类定义，而在该定义中又包含了 friend 函数 setX 的函数原型。
2. 一个执行文件（例如，Count.cpp），包含 Count 类成员函数的定义以及 friend 函数 setX 的定义。
3. 一个测试程序（例如，fig10_15.cpp），含有 main 函数。

使用非 friend 函数来修改 private 成员是错误的尝试

图 10.16 中的程序演示了当调用非 friend 函数 cannotSetX（第 29~32 行）修改 private 数据成员 x 时，编译器所产生的出错消息。

```
1  // Fig. 10.16: fig10_16.cpp
2  // Non-friend/non-member functions cannot access private data of a class.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // Count class definition (note that there is no friendship declaration)
8  class Count
9  {
10 public:
11    // constructor
12    Count()
13       : x( 0 ) // initialize x to 0
14    {
```

图 10.16　非 friend/ 非成员函数不能访问 private 成员

```
      // empty body
   } // end constructor Count

   // output x
   void print() const
   {
      cout << x << endl;
   } // end function print
private:
   int x; // data member
}; // end class Count

// function cannotSetX tries to modify private data of Count,
// but cannot because the function is not a friend of Count
void cannotSetX( Count &c, int val )
{
   c.x = val; // ERROR: cannot access private member in Count
} // end function cannotSetX

int main()
{
   Count counter; // create Count object

   cannotSetX( counter, 3 ); // cannotSetX is not a friend
   return 0;
} // end main
```

Borland C++ command-line compiler error message:

```
Error E2247 Fig10_16/fig10_16.cpp 31: 'Count::x' is not accessible in
   function cannotSetX(Count &,int)
```

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphtp5_examples\ch10\Fig10_16\fig10_16.cpp(31) : error C2248: 'Count::x'
   : cannot access private member declared in class 'Count'
        C:\scpphtp5_examples\ch10\Fig10_16\fig10_16.cpp(24) : see declaration
           of 'Count::x'
        C:\scpphtp5_examples\ch10\Fig10_16\fig10_16.cpp(9) : see declaration
           of 'Count'
```

GNU C++ compiler error messages:

```
fig10_16.cpp:24: error: `int Count::x' is private
fig10_16.cpp:31: error: within this context
```

图 10.16　非 friend/ 非成员函数不能访问 private 成员（续）

可以指定重载函数为类的友元。每个打算成为友元的重载函数必须在类的定义里显式地声明为类的一个友元。

10.5　使用 this 指针

我们已经看到对象的成员函数可以操作对象的数据。那么，成员函数如何知道哪个对象的数据成员要被操作呢？每个对象都可以使用一个称为this（一个C++保留字）的指针来访问自己的地址。对象的this指针不是对象本身的一部分 —— 也就是说，this指针占用的内存大小不会反映在对对象进行sizeof运算得到的结果中。相反，this指针作为一个隐式的参数（被编译器）传递给对象的每个非static成员函数。10.7 节将介绍 static 类成员，并解释为什么 this 指针不被隐式地传递给 static 成员函数。

对象隐式地使用this指针（就像我们目前所做的）或者显式地使用this指针来引用它们的数据成员和成员函数。this 指针的类型取决于对象的类型以及使用 this 的成员函数是否被声明为 const。

例如，在Employee类的非const成员函数中，this指针具有的类型是Employee * const（一个指向非const Employee对象的const指针）。可是在Employee类的const成员函数中，this指针具有的类型却为const Employee * const（指向一个const Employee对象的const指针）。

这一节我们的第一个例子展示了this指针隐式的和显式的用法，在本章稍后部分和第11章中，我们将给出几个真正巧妙地使用this的例子。

隐式和显式使用this指针来访问对象的数据成员

图10.17演示了隐式和显式使用this指针，使得Test类的成员函数可以打印Test对象的private数据x的情况。

```
1  // Fig. 10.17: fig10_17.cpp
2  // Using the this pointer to refer to object members.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  class Test
8  {
9  public:
10    Test( int = 0 ); // default constructor
11    void print() const;
12 private:
13    int x;
14 }; // end class Test
15
16 // constructor
17 Test::Test( int value )
18    : x( value ) // initialize x to value
19 {
20    // empty body
21 } // end constructor Test
22
23 // print x using implicit and explicit this pointers;
24 // the parentheses around *this are required
25 void Test::print() const
26 {
27    // implicitly use the this pointer to access the member x
28    cout << "        x = " << x;
29
30    // explicitly use the this pointer and the arrow operator
31    // to access the member x
32    cout << "\n  this->x = " << this->x;
33
34    // explicitly use the dereferenced this pointer and
35    // the dot operator to access the member x
36    cout << "\n(*this).x = " << ( *this ).x << endl;
37 } // end function print
38
39 int main()
40 {
41    Test testObject( 12 ); // instantiate and initialize testObject
42
43    testObject.print();
44    return 0;
45 } // end main
```

```
        x = 12
  this->x = 12
(*this).x = 12
```

图10.17　隐式和显式使用this指针来访问对象的成员

出于举例说明的目的，成员函数print（第25~37行）首先隐式地使用this指针打印x（第28行）——仅仅指明该数据成员的名称，然后 print 使用两种不同的表示法通过 this 指针访问 x ——一种是箭头运算符（->）紧跟着 this 指针（第 32 行），另一种是圆点运算符（.）紧跟着间接引用的 this 指针（第 36 行）。

请注意当 *this 与圆点成员选择运算符（.）一起使用时，括住 *this 的圆括号（第 36 行）。这对圆括号是必需的，因为圆点运算符具有比 * 运算符更高的优先级。如果不使用这对圆括号，表达式 *this.x 将被认为与有圆括号的 *（this.x）是相同的，进行求值会导致编译错误，因为圆点运算符不能与指针一起使用。

this指针的一个有趣用法是防止对象进行自我赋值。我们将在第11章看到，当对象包含指向动态分配内存的指针时，自我赋值可以导致严重错误。



常见的编程错误 10.7

试图将圆点成员选择运算符（.）和指向对象的指针一起使用将导致一个编译错误——圆点成员选择运算符只能与诸如对象名称、对象的引用或间接引用的指向对象的指针等左值一起使用。

使用 this 指针使串联的函数调用成为可能

this指针的另一种用法是使串联的成员函数调用成为可能，也就是多个函数在同一条语句中被调用（就像图10.20中的第14行一样）。图10.18至图10.20中的程序修改Time类的设置函数setTime、setHour、setMinute和setSecond，使得每个函数都返回对Time对象的引用，以便进行串联的成员函数调用。请注意在图 10.19 中上述的每个成员函数都在其函数体的最后一句语句返回 *this（第 26、33、40 和 47 行），返回类型是 Time &。

```
1  // Fig. 10.18: Time.h
2  // Cascading member function calls.
3
4  // Time class definition.
5  // Member functions defined in Time.cpp.
6  #ifndef TIME_H
7  #define TIME_H
8
9  class Time
10 {
11 public:
12    Time( int = 0, int = 0, int = 0 ); // default constructor
13
14    // set functions (the Time & return types enable cascading)
15    Time &setTime( int, int, int ); // set hour, minute, second
16    Time &setHour( int ); // set hour
17    Time &setMinute( int ); // set minute
18    Time &setSecond( int ); // set second
19
20    // get functions (normally declared const)
21    int getHour() const; // return hour
22    int getMinute() const; // return minute
23    int getSecond() const; // return second
24
25    // print functions (normally declared const)
26    void printUniversal() const; // print universal time
27    void printStandard() const; // print standard time
28 private:
29    int hour; // 0 - 23 (24-hour clock format)
30    int minute; // 0 - 59
31    int second; // 0 - 59
32 }; // end class Time
33
34 #endif
```

图 10.18　修改 Time 类的定义以进行串联的成员函数调用

```
1  // Fig. 10.19: Time.cpp
2  // Member-function definitions for Time class.
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setfill;
8  using std::setw;
9
10 #include "Time.h" // Time class definition
11
12 // constructor function to initialize private data;
13 // calls member function setTime to set variables;
14 // default values are 0 (see class definition)
15 Time::Time( int hr, int min, int sec )
16 {
17    setTime( hr, min, sec );
18 } // end Time constructor
19
20 // set values of hour, minute, and second
21 Time &Time::setTime( int h, int m, int s ) // note Time & return
22 {
23    setHour( h );
24    setMinute( m );
25    setSecond( s );
26    return *this; // enables cascading
27 } // end function setTime
28
29 // set hour value
30 Time &Time::setHour( int h ) // note Time & return
31 {
32    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
33    return *this; // enables cascading
34 } // end function setHour
35
36 // set minute value
37 Time &Time::setMinute( int m ) // note Time & return
38 {
39    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
40    return *this; // enables cascading
41 } // end function setMinute
42
43 // set second value
44 Time &Time::setSecond( int s ) // note Time & return
45 {
46    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
47    return *this; // enables cascading
48 } // end function setSecond
49
50 // get hour value
51 int Time::getHour() const
52 {
53    return hour;
54 } // end function getHour
55
56 // get minute value
57 int Time::getMinute() const
58 {
59    return minute;
60 } // end function getMinute
61
62 // get second value
```

图 10.19　修改 Time 类成员函数的定义以进行串联的成员函数调用

```
63 int Time::getSecond() const
64 {
65    return second;
66 } // end function getSecond
67
68 // print Time in universal-time format (HH:MM:SS)
69 void Time::printUniversal() const
70 {
71    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
72       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
73 } // end function printUniversal
74
75 // print Time in standard-time format (HH:MM:SS AM or PM)
76 void Time::printStandard() const
77 {
78    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
79       << ":" << setfill( '0' ) << setw( 2 ) << minute
80       << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
81 } // end function printStandard
```

图 10.19　修改 Time 类成员函数的定义以进行串联的成员函数调用（续）

```
1  // Fig. 10.20: fig10_20.cpp
2  // Cascading member function calls with the this pointer.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Time.h" // Time class definition
8
9  int main()
10 {
11    Time t; // create Time object
12
13    // cascaded function calls
14    t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
15
16    // output time in universal and standard formats
17    cout << "Universal time: ";
18    t.printUniversal();
19
20    cout << "\nStandard time: ";
21    t.printStandard();
22
23    cout << "\n\nNew standard time: ";
24
25    // cascaded function calls
26    t.setTime( 20, 20, 20 ).printStandard();
27    cout << endl;
28    return 0;
29 } // end main
```

```
Universal time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

图 10.20　串联的成员函数调用

图 10.20 的程序首先创建 Time 类对象 t（第 11 行），然后在串联的成员函数调用（第 14 行和第 26 行）中使用 t。为什么将 *this 作为一个引用返回的方法就可以支持串联的成员函数调用呢？我们知道圆点运算符（.）的结合律是从左向右的，因此第 14 行首先求 t.setHour(18) 的值，然后返回对对象 t 的引用，作为此函数调用的值。于是，其余的表达式就解释如下：

```
1  // Fig. 10.21: Employee.h
2  // Employee class definition.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  class Employee
7  {
8  public:
9     Employee( const char * const, const char * const ); // constructor
10    ~Employee(); // destructor
11    const char *getFirstName() const; // return first name
12    const char *getLastName() const; // return last name
13
14    // static member function
15    static int getCount(); // return number of objects instantiated
16 private:
17    char *firstName;
18    char *lastName;
19
20    // static data
21    static int count; // number of objects instantiated
22 }; // end class Employee
23
24 #endif
```

图 10.21　使用 static 数据成员跟踪内存中 Employee 对象数量的 Employee 类的定义

```
1  // Fig. 10.22: Employee.cpp
2  // Member-function definitions for class Employee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // strlen and strcpy prototypes
8  using std::strlen;
9  using std::strcpy;
10
11 #include "Employee.h" // Employee class definition
12
13 // define and initialize static data member at file scope
14 int Employee::count = 0;
15
16 // define static member function that returns number of
17 // Employee objects instantiated (declared static in Employee.h)
18 int Employee::getCount()
19 {
20    return count;
21 } // end static function getCount
22
23 // constructor dynamically allocates space for first and last name and
24 // uses strcpy to copy first and last names into the object
25 Employee::Employee( const char * const first, const char * const last )
26 {
27    firstName = new char[ strlen( first ) + 1 ];
28    strcpy( firstName, first );
29
30    lastName = new char[ strlen( last ) + 1 ];
31    strcpy( lastName, last );
32
33    count++; // increment static count of employees
34
35    cout << "Employee constructor for " << firstName
```

图 10.22　Employee 类的成员函数的定义

```
36          << ' ' << lastName << " called." << endl;
37    } // end Employee constructor
38
39    // destructor deallocates dynamically allocated memory
40    Employee::~Employee()
41    {
42       cout << "~Employee() called for " << firstName
43          << ' ' << lastName << endl;
44
45       delete [] firstName; // release memory
46       delete [] lastName; // release memory
47
48       count--; // decrement static count of employees
49    } // end ~Employee destructor
50
51    // return first name of employee
52    const char *Employee::getFirstName() const
53    {
54       // const before return type prevents client from modifying
55       // private data; client should copy returned string before
56       // destructor deletes storage to prevent undefined pointer
57       return firstName;
58    } // end function getFirstName
59
60    // return last name of employee
61    const char *Employee::getLastName() const
62    {
63       // const before return type prevents client from modifying
64       // private data; client should copy returned string before
65       // destructor deletes storage to prevent undefined pointer
66       return lastName;
67    } // end function getLastName
```

图 10.22　Employee 类的成员函数的定义（续）

在图 10.22 中，注意在 Employee 构造函数中通过使用 new 运算符（第 27 和 30 行）来为成员 firstName 和 lastName 动态分配确切大小的内存空间。如果 new 运算符不能满足其中一个或者两个字符数组的内存空间需求，程序将立即终止。

在图 10.22 中还要注意的是，getFirstName 函数（第 52~58 行）和 getLastName 函数（第 61~67 行）的实现都返回了指向 const 字符数据的指针。在此实现中，如果客户希望保留一份姓或者名的拷贝，那么当客户从对象调用端获得指向 const 字符数据的指针后它负责复制 Employee 对象中动态分配的内存值。也有可能为了实现 getFirstName 和 getLastName，要求客户将一个字符数组及其大小传递给每个函数。然后函数就将姓或者名复制到客户提供的字符数组中。再次说明的是，这里我们还可以使用 string 类向调用者返回 string 对象的一份拷贝，而不是返回指向 private 数据的指针。

图 10.23 用 static 成员函数 getCount 确定当前实例化的 Employee 对象的数量。请注意，当程序中没有实例化的对象时，采用的是 Employee::getCount()函数调用（第 14 行和第 38 行）。但是，当有实例化的对象时，函数 getCount 可以通过这些对象中的任意一个来调用，如第 22~23 行中的语句所示，它使用 e1Ptr 指针调用函数 getCount。注意第 23 行中无论是使用 e2Ptr->getCount()还是 Employee::getCount()都将产生同样的结果，因为 getCount 函数总是访问同一个 static 成员 count。



软件工程知识 10.11

一些组织在他们的软件工程标准中规定所有对 static 成员函数的调用只能通过类名而不是对象句柄。

如果成员函数不访问类的非 static 的数据成员或非 static 的成员函数，那么它应当声明为 static。与非 static 成员函数不同的是，static 成员函数不具有 this 指针，因为 static 数据成员和 static 成员函

数独立于类的任何对象而存在。this指针必须指向类的具体的对象，当static成员函数被调用时，内存中也许没有类的任何对象存在。

```
// Fig. 10.23: fig10_23.cpp
// Driver to test class Employee.
#include <iostream>
using std::cout;
using std::endl;

#include "Employee.h" // Employee class definition

int main()
{
   // use class name and binary scope resolution operator to
   // access static number function getCount
   cout << "Number of employees before instantiation of any objects is "
      << Employee::getCount() << endl; // use class name

   // use new to dynamically create two new Employees
   // operator new also calls the object's constructor
   Employee *e1Ptr = new Employee( "Susan", "Baker" );
   Employee *e2Ptr = new Employee( "Robert", "Jones" );

   // call getCount on first Employee object
   cout << "Number of employees after objects are instantiated is "
      << e1Ptr->getCount();

   cout << "\n\nEmployee 1: "
      << e1Ptr->getFirstName() << " " << e1Ptr->getLastName()
      << "\nEmployee 2: "
      << e2Ptr->getFirstName() << " " << e2Ptr->getLastName() << "\n\n";

   delete e1Ptr; // deallocate memory
   e1Ptr = 0; // disconnect pointer from free-store space
   delete e2Ptr; // deallocate memory
   e2Ptr = 0; // disconnect pointer from free-store space

   // no objects exist, so call static member function getCount again
   // using the class name and the binary scope resolution operator
   cout << "Number of employees after objects are deleted is "
      << Employee::getCount() << endl;
   return 0;
} // end main
```

```
Number of employees before instantiation of any objects is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after objects are instantiated is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after objects are deleted is 0
```

图 10.23　跟踪类的对象数目的 static 数据成员



常见的编程错误 10.11

在 static 成员函数中使用 this 指针是一个编译错误。

介绍代理类（proxy class）的概念，即使是类的 private 数据，代理类也允许你向类的客户隐藏。通过向类的客户提供只知道类的 public 接口的代理类，就可以使客户能够使用类的服务，而又无法访问类的实现细节。

实现代理类需要几个步骤，我们在图 10.24 至图 10.27 中演示。首先，我们为包含欲隐藏的私有实现的类生成类定义，在例中该类称为 Implementation，如图 10.24 所示。在图 10.25 和图 10.26 中展示了代理类 Interface。测试程序和样例输出如图 10.27 所示。

类 Implementation（见图 10.24）提供了一个名为 value 的 private 数据成员（这正是我们想要向客户隐藏的数据）、一个初始化 value 的构造函数以及函数 setValue 和 getValue。

我们用与 Implementation 一样的 public 接口（除了构造函数和析构函数的名称不同外）定义了一个名为 Interface（见图 10.25）的代理类。

```
1  // Fig. 10.24: Implementation.h
2  // Header file for class Implementation
3
4  class Implementation
5  {
6  public:
7     // constructor
8     Implementation( int v )
9        : value( v ) // initialize value with v
10    {
11       // empty body
12    } // end constructor Implementation
13
14    // set value to v
15    void setValue( int v )
16    {
17       value = v; // should validate v
18    } // end function setValue
19
20    // return value
21    int getValue() const
22    {
23       return value;
24    } // end function getValue
25 private:
26    int value; // data that we would like to hide from the client
27 }; // end class Implementation
```

图 10.24　Implementation 类的定义

```
1  // Fig. 10.25: Interface.h
2  // Header file for class Interface
3  // Client sees this source code, but the source code does not reveal
4  // the data layout of class Implementation.
5
6  class Implementation; // forward class declaration required by line 17
7
8  class Interface
9  {
10 public:
11    Interface( int ); // constructor
12    void setValue( int ); // same public interface as
13    int getValue() const; // class Implementation has
14    ~Interface(); // destructor
15 private:
16    // requires previous forward declaration (line 6)
17    Implementation *ptr;
18 }; // end class Interface
```

图 10.25　Interface 类的定义

这个代理类惟一的private成员是一个指向Implementation类对象的指针。以这样的方式使用指针使我们能够向客户隐藏Implementation类的实现细节。请注意在Interface类里提到专有Implementation 类的地方只有两处：一处是在第 17 行指针的声明中，另一处是在第 6 行的提前类声明中（forward class declaration）。当类定义（如Interface类）仅使用指向另一个类的对象（如Implementation类的对象）的指针或引用时，另一个类的头文件（通常会泄露该类的private数据）不需要用#include包含进来。只需在文件中使用前面所述的另一个类之前，简单地用提前声明（第6行）将其声明为一种数据类型。

Interface代理类的成员函数实现文件（见图10.26）是惟一包含头文件Implementation.h（第5行）的文件，头文件 Implementation.h 包含 Implementation 类。文件 Interface.cpp（见图 10.26）以预编译后的目标代码文件的形式和头文件 Interface.h 一起提供给客户，其中头文件 Interface 包含代理类提供的服务的函数原型。因为文件 Interface.cpp 只以目标代码的形式提供给客户使用，所以客户无法看到代理类和专有类间的互相作用（第9，17，23 和 29 行）。请注意，代理类以强加的额外的函数调用“层”，作为隐藏 Implementation 类的 private 对象的“代价”。考虑到当今计算机的速度以及许多编译器可以自动内联简单的函数调用，这些额外的函数调用对性能的影响常常可以忽略不计。

```
1  // Fig. 10.26: Interface.cpp
2  // Implementation of class Interface--client receives this file only
3  // as precompiled object code, keeping the implementation hidden.
4  #include "Interface.h" // Interface class definition
5  #include "Implementation.h" // Implementation class definition
6
7  // constructor
8  Interface::Interface( int v )
9     : ptr ( new Implementation( v ) ) // initialize ptr to point to
10 {                                    // a new Implementation object
11    // empty body
12 } // end Interface constructor
13
14 // call Implementation's setValue function
15 void Interface::setValue( int v )
16 {
17    ptr->setValue( v );
18 } // end function setValue
19
20 // call Implementation's getValue function
21 int Interface::getValue() const
22 {
23    return ptr->getValue();
24 } // end function getValue
25
26 // destructor
27 Interface::~Interface()
28 {
29    delete ptr;
30 } // end ~Interface destructor
```

图 10.26　Interface 类成员函数的定义

图 10.27 测试 Interface 类。注意，只有 Interface 的头文件包含在客户代码（第 7 行）中——并未提及一个独立的称为 Implementation 类的存在。因此，客户永远不会看到 Implementation 类的 private 数据，客户代码也不能依赖 Implementation 代码。



软件工程知识 10.13
代理类将客户代码与实现的变化相隔离。

```
// Fig. 10.27: fig10_27.cpp
// Hiding a class's private data with a proxy class.
#include <iostream>
using std::cout;
using std::endl;

#include "Interface.h" // Interface class definition

int main()
{
   Interface i( 5 ); // create Interface object

   cout << "Interface contains: " << i.getValue()
      << " before setValue" << endl;

   i.setValue( 10 );

   cout << "Interface contains: " << i.getValue()
      << " after setValue" << endl;
   return 0;
} // end main
```

```
Interface contains: 5 before setValue
Interface contains: 10 after setValue
```

图10.27　实现代理类

10.11　本章小结

在这一章里，我们介绍了许多关于类和数据抽象的高级话题。学习了如何指定const对象和const成员函数以防止对对象进行修改，从而增强最小特权原则。还学习了通过组成，类可以使其他类的对象作为其成员。我们介绍了友元关系并且给出了例子演示如何使用friend函数。

读者明白了this指针会作为隐含的参数传递给类的每个非static成员函数，使得这些函数可以访问当前对象的数据成员和其他的非static成员函数。也看到了显式使用this指针可以访问类的成员和串联地进行成员函数调用。

本章介绍了动态内存管理的概念。读者知道了C++程序员可以使用new和delete运算符动态创建和撤销对象。我们激发了对static数据成员的需求并演示了如何在自己的类中声明和使用static数据成员和成员函数。

读者学习了数据抽象和信息隐藏这两个面向对象程序设计的基本概念。我们讨论了抽象数据类型——在计算机系统内用可以接受的精度表达真实世界或是理性概念的方法。然后学习了三个抽象数据类型的例子——数组、字符串和队列。我们介绍了用来存储一系列对象的容器类的概念，以及用来遍历容器内元素的迭代器的概念。最后，学习了如何建立代理类以向类的客户隐藏类的实现细节（包括private数据成员）。

在第11章里，我们将通过介绍如何使C++的运算符能够和对象一起工作——一种称为运算符重载的方法，继续类和对象的学习。例如，读者将看到如何“重载”<<运算符，使它不需要显式循环语句的支持，就可以用来输出完整的数组。

摘要

- 关键字const可用来指定对象是不可更改的并且任何企图修改对象的操作都将导致一个编译错误。
- C++编译器不允许const对象调用非const成员函数。

输入 PhoneNumber 类的对象中。当编译器遇到图 11.5 中第 19 行的表达式：

```
cin >> phone
```

时，它就会产生如下的全局函数调用：

```
operator>>( cin, phone );
```

```
1  // Fig. 11.3: PhoneNumber.h
2  // PhoneNumber class definition
3  #ifndef PHONENUMBER_H
4  #define PHONENUMBER_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 #include <string>
11 using std::string;
12
13 class PhoneNumber
14 {
15    friend ostream &operator<<( ostream &, const PhoneNumber & );
16    friend istream &operator>>( istream &, PhoneNumber & );
17 private:
18    string areaCode; // 3-digit area code
19    string exchange; // 3-digit exchange
20    string line; // 4-digit line
21 }; // end class PhoneNumber
22
23 #endif
```

图 11.3　以重载的流插入和流提取运算符作为友元函数的 PhoneNumber 类

```
1  // Fig. 11.4: PhoneNumber.cpp
2  // Overloaded stream insertion and stream extraction operators
3  // for class PhoneNumber.
4  #include <iomanip>
5  using std::setw;
6
7  #include "PhoneNumber.h"
8
9  // overloaded stream insertion operator; cannot be
10 // a member function if we would like to invoke it with
11 // cout << somePhoneNumber;
12 ostream &operator<<( ostream &output, const PhoneNumber &number )
13 {
14    output << "(" << number.areaCode << ") "
15       << number.exchange << "-" << number.line;
16    return output; // enables cout << a << b << c;
17 } // end function operator<<
18
19 // overloaded stream extraction operator; cannot be
20 // a member function if we would like to invoke it with
21 // cin >> somePhoneNumber;
22 istream &operator>>( istream &input, PhoneNumber &number )
23 {
24    input.ignore(); // skip (
25    input >> setw( 3 ) >> number.areaCode; // input area code
26    input.ignore( 2 ); // skip ) and space
27    input >> setw( 3 ) >> number.exchange; // input exchange
28    input.ignore(); // skip dash (-)
29    input >> setw( 4 ) >> number.line; // input line
30    return input; // enables cin >> a >> b >> c;
31 } // end function operator>>
```

图 11.4　为类 PhoneNumber 重载流插入和流提取运算符

```
1  // Fig. 11.5: fig11_05.cpp
2  // Demonstrating class PhoneNumber's overloaded stream insertion
3  // and stream extraction operators.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include "PhoneNumber.h"
10
11 int main()
12 {
13    PhoneNumber phone; // create object phone
14
15    cout << "Enter phone number in the form (123) 456-7890:" << endl;
16
17    // cin >> phone invokes operator>> by implicitly issuing
18    // the global function call operator>>( cin, phone )
19    cin >> phone;
20
21    cout << "The phone number entered was: ";
22
23    // cout << phone invokes operator<< by implicitly issuing
24    // the global function call operator<<( cout, phone )
25    cout << phone << endl;
26    return 0;
27 } // end main
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

图 11.5　重载的流插入和流提取运算符

当这个函数调用执行时，引用形参 input（见图 11.4 中的第 22 行）成为 cin 的别名，而引用形参 number 成为 phone 的别名。运算符函数把电话号码的 3 个部分作为字符串分别读到被形参 number 引用的 PhoneNumber 对象的成员变量 areaCode（第 25 行）、exchange（第 27 行）和 line（第 29 行）中。流操纵元 setw 限定了读到每个字符数组的字符个数。当和 cin 及字符串一起使用时，setw 把读入的字符个数限定为其参数指定的字符个数，即 setw(3)允许读入 3 个字符。通过调用 istream 的成员函数 ignore（见图 11.4 中的第 24、26 和 28 行），圆括号、空格和破折号等字符都被跳过。函数 ignore 丢弃输入流中指定个数的字符（默认为一个字符）。函数 operator>>返回 istream 引用 input（即 cin）。这样，使 PhoneNumber 对象上的输入操作可以和其他 PhoneNumber 对象或者其他数据类型对象上的输入操作串联起来。例如，程序可以在如下的语句中输入两个 PhoneNumber 对象：

```
cin >> phone1 >> phone2;
```

首先，通过下面的全局函数调用执行表达式 cin >> phone1：

```
operator>>( cin, phone1 );
```

这个函数调用返回 cin 的引用，并将它作为 cin >> phone1 的值，因此表达式的剩余部分可以简单理解成 cin >> phone2，这将通过下列的全局函数调用得以执行：

```
1  // Fig. 11.6: Array.h
2  // Array class for storing arrays of integers.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 class Array
11 {
12    friend ostream &operator<<( ostream &, const Array & );
13    friend istream &operator>>( istream &, Array & );
14 public:
15    Array( int = 10 ); // default constructor
16    Array( const Array & ); // copy constructor
17    ~Array(); // destructor
18    int getSize() const; // return size
19
20    const Array &operator=( const Array & ); // assignment operator
21    bool operator==( const Array & ) const; // equality operator
22
23    // inequality operator; returns opposite of == operator
24    bool operator!=( const Array &right ) const
25    {
26       return ! ( *this == right ); // invokes Array::operator==
27    } // end function operator!=
28
29    // subscript operator for non-const objects returns modifiable lvalue
30    int &operator[]( int );
31
32    // subscript operator for const objects returns rvalue
33    int operator[]( int ) const;
34 private:
35    int size; // pointer-based array size
36    int *ptr; // pointer to first element of pointer-based array
37 }; // end class Array
38
39 #endif
```

图 11.6　具有重载运算符的 Array 类的定义

```
1  // Fig 11.7: Array.cpp
2  // Member-function definitions for class Array
3  #include <iostream>
4  using std::cerr;
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // exit function prototype
13 using std::exit;
14
15 #include "Array.h" // Array class definition
16
17 // default constructor for class Array (default size 10)
18 Array::Array( int arraySize )
19 {
```

图 11.7　Array 类成员函数和友元函数的定义

```
20     size = ( arraySize > 0 ? arraySize : 10 ); // validate arraySize
21     ptr = new int[ size ]; // create space for pointer-based array
22
23     for ( int i = 0; i < size; i++ )
24        ptr[ i ] = 0; // set pointer-based array element
25  } // end Array default constructor
26
27  // copy constructor for class Array;
28  // must receive a reference to prevent infinite recursion
29  Array::Array( const Array &arrayToCopy )
30     : size( arrayToCopy.size )
31  {
32     ptr = new int[ size ]; // create space for pointer-based array
33
34     for ( int i = 0; i < size; i++ )
35        ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
36  } // end Array copy constructor
37
38  // destructor for class Array
39  Array::~Array()
40  {
41     delete [] ptr; // release pointer-based array space
42  } // end destructor
43
44  // return number of elements of Array
45  int Array::getSize() const
46  {
47     return size; // number of elements in Array
48  } // end function getSize
49
50  // overloaded assignment operator;
51  // const return avoids: ( a1 = a2 ) = a3
52  const Array &Array::operator=( const Array &right )
53  {
54     if ( &right != this ) // avoid self-assignment
55     {
56        // for Arrays of different sizes, deallocate original
57        // left-side array, then allocate new left-side array
58        if ( size != right.size )
59        {
60           delete [] ptr; // release space
61           size = right.size; // resize this object
62           ptr = new int[ size ]; // create space for array copy
63        } // end inner if
64
65        for ( int i = 0; i < size; i++ )
66           ptr[ i ] = right.ptr[ i ]; // copy array into object
67     } // end outer if
68
69     return *this; // enables x = y = z, for example
70  } // end function operator=
71
72  // determine if two Arrays are equal and
73  // return true, otherwise return false
74  bool Array::operator==( const Array &right ) const
75  {
76     if ( size != right.size )
77        return false; // arrays of different number of elements
78
79     for ( int i = 0; i < size; i++ )
80        if ( ptr[ i ] != right.ptr[ i ] )
81           return false; // Array contents are not equal
```

图 11.7　Array 类成员函数和友元函数的定义（续）

```
82
83    return true; // Arrays are equal
84 } // end function operator==
85
86 // overloaded subscript operator for non-const Arrays;
87 // reference return creates a modifiable lvalue
88 int &Array::operator[]( int subscript )
89 {
90    // check for subscript out-of-range error
91    if ( subscript < 0 || subscript >= size )
92    {
93       cerr << "\nError: Subscript " << subscript
94          << " out of range" << endl;
95       exit( 1 ); // terminate program; subscript out of range
96    } // end if
97
98    return ptr[ subscript ]; // reference return
99 } // end function operator[]
100
101 // overloaded subscript operator for const Arrays
102 // const reference return creates an rvalue
103 int Array::operator[]( int subscript ) const
104 {
105    // check for subscript out-of-range error
106    if ( subscript < 0 || subscript >= size )
107    {
108       cerr << "\nError: Subscript " << subscript
109          << " out of range" << endl;
110       exit( 1 ); // terminate program; subscript out of range
111    } // end if
112
113    return ptr[ subscript ]; // returns copy of this element
114 } // end function operator[]
115
116 // overloaded input operator for class Array;
117 // inputs values for entire Array
118 istream &operator>>( istream &input, Array &a )
119 {
120    for ( int i = 0; i < a.size; i++ )
121       input >> a.ptr[ i ];
122
123    return input; // enables cin >> x >> y;
124 } // end function
125
126 // overloaded output operator for class Array
127 ostream &operator<<( ostream &output, const Array &a )
128 {
129    int i;
130
131    // output private ptr-based array
132    for ( i = 0; i < a.size; i++ )
133    {
134       output << setw( 12 ) << a.ptr[ i ];
135
136       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
137          output << endl;
138    } // end for
139
140    if ( i % 4 != 0 ) // end last line of output
141       output << endl;
142
143    return output; // enables cout << x << y;
144 } // end function operator<<
```

图 11.7　Array 类成员函数和友元函数的定义（续）

```
1  // Fig. 11.8: fig11_08.cpp
2  // Array class test program.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include "Array.h"
9
10 int main()
11 {
12    Array integers1( 7 ); // seven-element Array
13    Array integers2; // 10-element Array by default
14
15    // print integers1 size and contents
16    cout << "Size of Array integers1 is "
17       << integers1.getSize()
18       << "\nArray after initialization:\n" << integers1;
19
20    // print integers2 size and contents
21    cout << "\nSize of Array integers2 is "
22       << integers2.getSize()
23       << "\nArray after initialization:\n" << integers2;
24
25    // input and print integers1 and integers2
26    cout << "\nEnter 17 integers:" << endl;
27    cin >> integers1 >> integers2;
28
29    cout << "\nAfter input, the Arrays contain:\n"
30       << "integers1:\n" << integers1
31       << "integers2:\n" << integers2;
32
33    // use overloaded inequality (!=) operator
34    cout << "\nEvaluating: integers1 != integers2" << endl;
35
36    if ( integers1 != integers2 )
37       cout << "integers1 and integers2 are not equal" << endl;
38
39    // create Array integers3 using integers1 as an
40    // initializer; print size and contents
41    Array integers3( integers1 ); // invokes copy constructor
42
43    cout << "\nSize of Array integers3 is "
44       << integers3.getSize()
45       << "\nArray after initialization:\n" << integers3;
46
47    // use overloaded assignment (=) operator
48    cout << "\nAssigning integers2 to integers1:" << endl;
49    integers1 = integers2; // note target Array is smaller
50
51    cout << "integers1:\n" << integers1
52       << "integers2:\n" << integers2;
53
54    // use overloaded equality (==) operator
55    cout << "\nEvaluating: integers1 == integers2" << endl;
56
57    if ( integers1 == integers2 )
58       cout << "integers1 and integers2 are equal" << endl;
59
60    // use overloaded subscript operator to create rvalue
61    cout << "\nintegers1[5] is " << integers1[ 5 ];
62
```

图 11.8　Array 类测试程序

```
63      // use overloaded subscript operator to create lvalue
64      cout << "\n\nAssigning 1000 to integers1[5]" << endl;
65      integers1[ 5 ] = 1000;
66      cout << "integers1:\n" << integers1;
67
68      // attempt to use out-of-range subscript
69      cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
70      integers1[ 15 ] = 1000; // ERROR: out of range
71      return 0;
72   } // end main
```

```
Size of Array integers1 is 7
Array after initialization:
           0           0           0           0
           0           0           0

Size of Array integers2 is 10
Array after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the Arrays contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of Array integers3 is 7
Array after initialization:
           1           2           3           4
           5           6           7

Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range
```

图 11.8　Array 类测试程序（续）

声明了两个重载的强制类型转换运算符函数，它们分别可以将用户自定义类型A的对象转换成整数和用户自定义类型 OtherClass 的对象。

强制类型转换运算符和转换构造函数的优点之一就是：必要时，编译器可以隐式地调用这些函数来创建临时的对象。例如，如果用户自定义的 String 类的对象 s 出现在程序中一个本该是出现普通的 char * 数据的位置上，如下所示：

```
cout << s;
```

那么编译器便可以调用重载的强制类型转换运算符函数 operator char *，将对象 s 转换成 char *，并在表达式中使用这个转换结果char *。如果为我们的String类提供这个强制类型转换运算符的话，就不必重载流插入运算符用 cout 输出 String 对象了。

11.10 实例研究：String 类

作为学习重载的最佳练习，我们将构建自己的 String 类来处理字符串的创建和操作（见图 11.9 到图 11.11）。C++标准库也提供了与之类似、但更为强大的类 string。回忆一下，这个类曾在第 3 章中介绍过，在 11.13 节我们还会给出使用它的另外的示例。现在，我们将充分利用运算符重载来打造我们自己的 String 类。

首先，我们介绍String类的头文件，讨论用于表示String对象的私有数据。然后浏览类的public接口，讨论 String 类提供的每种服务。我们将讨论 String 类的成员函数定义。对每个重载的运算符函数，显示在程序中调用它们的代码，并解释这些函数是如何工作的。

String 类的定义

现在让我们通读一下图 11.9 中 String 类的头文件，首先看一看 String 的基于指针的内部表示。第 55~56 行声明了 String 类的 private 数据成员。该类有一个 length 域，表示字符串中的字符个数，不包括结尾的空字符；还有一个指针 sPtr，指向代表字符串的动态分配的内存。

```
1  // Fig. 11.9: String.h
2  // String class definition.
3  #ifndef STRING_H
4  #define STRING_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 class String
11 {
12    friend ostream &operator<<( ostream &, const String & );
13    friend istream &operator>>( istream &, String & );
14 public:
15    String( const char * = "" ); // conversion/default constructor
16    String( const String & ); // copy constructor
17    ~String(); // destructor
18
19    const String &operator=( const String & ); // assignment operator
20    const String &operator+=( const String & ); // concatenation operator
21
22    bool operator!() const; // is String empty?
23    bool operator==( const String & ) const; // test s1 == s2
24    bool operator<( const String & ) const; // test s1 < s2
```

图 11.9 具有运算符重载的 String 类的定义

```
25
26    // test s1 != s2
27    bool operator!=( const String &right ) const
28    {
29       return !( *this == right );
30    } // end function operator!=
31
32    // test s1 > s2
33    bool operator>( const String &right ) const
34    {
35       return right < *this;
36    } // end function operator>
37
38    // test s1 <= s2
39    bool operator<=( const String &right ) const
40    {
41       return !( right < *this );
42    } // end function operator <=
43
44    // test s1 >= s2
45    bool operator>=( const String &right ) const
46    {
47       return !( *this < right );
48    } // end function operator>=
49
50    char &operator[]( int ); // subscript operator (modifiable lvalue)
51    char operator[]( int ) const; // subscript operator (rvalue)
52    String operator()( int, int = 0 ) const; // return a substring
53    int getLength() const; // return string length
54 private:
55    int length; // string length (not counting null terminator)
56    char *sPtr; // pointer to start of pointer-based string
57
58    void setString( const char * ); // utility function
59 }; // end class String
60
61 #endif
```

图 11.9　具有运算符重载的 String 类的定义（续）

```
1  // Fig. 11.10: String.cpp
2  // Member-function definitions for class String.
3  #include <iostream>
4  using std::cerr;
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 #include <cstring> // strcpy and strcat prototypes
12 using std::strcmp;
13 using std::strcpy;
14 using std::strcat;
15
16 #include <cstdlib> // exit prototype
17 using std::exit;
18
19 #include "String.h" // String class definition
20
21 // conversion (and default) constructor converts char * to String
22 String::String( const char *s )
23    : length( ( s != 0 ) ? strlen( s ) : 0 )
```

图 11.10　String 类成员函数和友元函数的定义

```
{
   cout << "Conversion (and default) constructor: " << s << endl;
   setString( s ); // call utility function
} // end String conversion constructor

// copy constructor
String::String( const String &copy )
   : length( copy.length )
{
   cout << "Copy constructor: " << copy.sPtr << endl;
   setString( copy.sPtr ); // call utility function
} // end String copy constructor

// Destructor
String::~String()
{
   cout << "Destructor: " << sPtr << endl;
   delete [] sPtr; // release pointer-based string memory
} // end ~String destructor

// overloaded = operator; avoids self assignment
const String &String::operator=( const String &right )
{
   cout << "operator= called" << endl;

   if ( &right != this ) // avoid self assignment
   {
      delete [] sPtr; // prevents memory leak
      length = right.length; // new String length
      setString( right.sPtr ); // call utility function
   } // end if
   else
      cout << "Attempted assignment of a String to itself" << endl;

   return *this; // enables cascaded assignments
} // end function operator=

// concatenate right operand to this object and store in this object
const String &String::operator+=( const String &right )
{
   size_t newLength = length + right.length; // new length
   char *tempPtr = new char[ newLength + 1 ]; // create memory

   strcpy( tempPtr, sPtr ); // copy sPtr
   strcpy( tempPtr + length, right.sPtr ); // copy right.sPtr

   delete [] sPtr; // reclaim old space
   sPtr = tempPtr; // assign new array to sPtr
   length = newLength; // assign new length to length
   return *this; // enables cascaded calls
} // end function operator+=

// is this String empty?
bool String::operator!() const
{
   return length == 0;
} // end function operator!

// Is this String equal to right String?
bool String::operator==( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) == 0;
```

图 11.10　String 类成员函数和友元函数的定义（续）

```
} // end function operator==

// Is this String less than right String?
bool String::operator<( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) < 0;
} // end function operator<

// return reference to character in String as a modifiable lvalue
char &String::operator[]( int subscript )
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length )
   {
      cerr << "Error: Subscript " << subscript
         << " out of range" << endl;
      exit( 1 ); // terminate program
   } // end if

   return sPtr[ subscript ]; // non-const return; modifiable lvalue
} // end function operator[]

// return reference to character in String as rvalue
char String::operator[]( int subscript ) const
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length )
   {
      cerr << "Error: Subscript " << subscript
         << " out of range" << endl;
      exit( 1 ); // terminate program
   } // end if

   return sPtr[ subscript ]; // returns copy of this element
} // end function operator[]

// return a substring beginning at index and of length subLength
String String::operator()( int index, int subLength ) const
{
   // if index is out of range or substring length < 0,
   // return an empty String object
   if ( index < 0 || index >= length || subLength < 0 )
      return ""; // converted to a String object automatically

   // determine length of substring
   int len;

   if ( ( subLength == 0 ) || ( index + subLength > length ) )
      len = length - index;
   else
      len = subLength;

   // allocate temporary array for substring and
   // terminating null character
   char *tempPtr = new char[ len + 1 ];

   // copy substring into char array and terminate string
   strncpy( tempPtr, &sPtr[ index ], len );
   tempPtr[ len ] = '\0';

   // create temporary String object containing the substring
   String tempString( tempPtr );
```

图 11.10　String 类成员函数和友元函数的定义（续）

```
148    delete [] tempPtr; // delete temporary array
149    return tempString; // return copy of the temporary String
150 } // end function operator()
151
152 // return string length
153 int String::getLength() const
154 {
155    return length;
156 } // end function getLength
157
158 // utility function called by constructors and operator=
159 void String::setString( const char *string2 )
160 {
161    sPtr = new char[ length + 1 ]; // allocate memory
162
163    if ( string2 != 0 ) // if string2 is not null pointer, copy contents
164       strcpy( sPtr, string2 ); // copy literal to object
165    else // if string2 is a null pointer, make this an empty string
166       sPtr[ 0 ] = '\0'; // empty string
167 } // end function setString
168
169 // overloaded output operator
170 ostream &operator<<( ostream &output, const String &s )
171 {
172    output << s.sPtr;
173    return output; // enables cascading
174 } // end function operator<<
175
176 // overloaded input operator
177 istream &operator>>( istream &input, String &s )
178 {
179    char temp[ 100 ]; // buffer to store input
180    input >> setw( 100 ) >> temp;
181    s = temp; // use String class assignment operator
182    return input; // enables cascading
183 } // end function operator>>
```

图 11.10　String 类成员函数和友元函数的定义（续）

```
 1 // Fig. 11.11: fig11_11.cpp
 2 // String class test program.
 3 #include <iostream>
 4 using std::cout;
 5 using std::endl;
 6 using std::boolalpha;
 7
 8 #include "String.h"
 9
10 int main()
11 {
12    String s1( "happy" );
13    String s2( " birthday" );
14    String s3;
15
16    // test overloaded equality and relational operators
17    cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
18       << "\"; s3 is \"" << s3 << '\"'
19       << boolalpha << "\n\nThe results of comparing s2 and s1:"
20       << "\ns2 == s1 yields " << ( s2 == s1 )
21       << "\ns2 != s1 yields " << ( s2 != s1 )
```

图 11.11　String 类的测试程序

```
      << "\ns2 >  s1 yields " << ( s2 > s1 )
      << "\ns2 <  s1 yields " << ( s2 < s1 )
      << "\ns2 >= s1 yields " << ( s2 >= s1 )
      << "\ns2 <= s1 yields " << ( s2 <= s1 );

   // test overloaded String empty (!) operator
   cout << "\n\nTesting !s3:" << endl;

   if ( !s3 )
   {
      cout << "s3 is empty; assigning s1 to s3;" << endl;
      s3 = s1; // test overloaded assignment
      cout << "s3 is \"" << s3 << "\"";
   } // end if

   // test overloaded String concatenation operator
   cout << "\n\ns1 += s2 yields s1 = ";
   s1 += s2; // test overloaded concatenation
   cout << s1;

   // test conversion constructor
   cout << "\n\ns1 += \" to you\" yields" << endl;
   s1 += " to you"; // test conversion constructor
   cout << "s1 = " << s1 << "\n\n";

   // test overloaded function call operator () for substring
   cout << "The substring of s1 starting at\n"
      << "location 0 for 14 characters, s1(0, 14), is:\n"
      << s1( 0, 14 ) << "\n\n";

   // test substring "to-end-of-String" option
   cout << "The substring of s1 starting at\n"
      << "location 15, s1(15), is: "
      << s1( 15 ) << "\n\n";

   // test copy constructor
   String *s4Ptr = new String( s1 );
   cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";

   // test assignment (=) operator with self-assignment
   cout << "assigning *s4Ptr to *s4Ptr" << endl;
   *s4Ptr = *s4Ptr; // test overloaded assignment
   cout << "*s4Ptr = " << *s4Ptr << endl;

   // test destructor
   delete s4Ptr;

   // test using subscript operator to create a modifiable lvalue
   s1[ 0 ] = 'H';
   s1[ 6 ] = 'B';
   cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
      << s1 << "\n\n";

   // test subscript out of range
   cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
   s1[ 30 ] = 'd'; // ERROR: subscript out of range
   return 0;
} // end main
```

图 11.11　String 类的测试程序（续）

```
Conversion (and default) constructor: happy
Conversion (and default) constructor:  birthday
Conversion (and default) constructor:
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion (and default) constructor:  to you
Destructor:  to you
s1 = happy birthday to you

Conversion (and default) constructor: happy birthday
Copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday

Destructor: happy birthday
Conversion (and default) constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15), is: to you

Destructor: to you
Copy constructor: happy birthday to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself

*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
Error: Subscript 30 out of range
```

图 11.11　String 类的测试程序（续）

将流插入运算符和流提取运算符重载成友元函数

图 11.9 中的第 12~13 行将重载的流插入运算符函数 operator<<（定义在图 11.10 中的第 170~174 行）和重载的流提取运算符函数 operator>>（定义在图 11.10 中的第 177~183 行）声明为 String 类的友元。operator<<的实现直截了当。请注意，operator>>采用 setw 将可读入数组 temp 的字符总数限定在 99 个（第 180 行），而第 100 个字符位置是专门为字符串的终止空字符保留的。注意：

```
Date operator++( int )
```

实参 0 纯粹是个“哑值”，它使编译器能够区分前置的和后置的自增运算符函数。

如果以全局函数实现后置的自增运算，那么当编译器遇到表达式d1++时，会产生如下的函数调用：

```
operator++( d1, 0 )
```

这个函数的原型应是：

```
Date operator++( Date &, int );
```

同样，这里的参数0是编译器用来区分以全局函数实现的前置和后置自增运算符的。请注意，后置的自增运算符按值返回 Date 对象，而前置的自增运算符按引用返回 Date 对象。这是因为在进行自增前，后置的自增运算符通常先返回一个包含对象原始值的临时对象。C++将这样的对象作为右值处理，使其不能用在赋值运算符的左侧。前置的自增运算符返回实际自增后的具有新值的对象。这种对象在连续的表达式中可以作为左值使用。



性能提示 11.3

由后置的自增(或自减)运算符创建的临时对象能对性能造成很大的影响——尤其是在循环中使用这个运算符时。出于这个原因，仅当程序的逻辑要求后置自增（或后置自减）操作时，才应该使用后置自增（或自减）运算符。

本节关于重载前置的和后置的自增运算符的一切叙述对重载前置的自减和后置的自减运算符同样适用。接下来，我们将分析使用重载的前置和后置自增运算符的 Date 类。

11.12　实例研究：Date 类

图11.12到图11.14中的程序演示了Date类。这个类用重载的前置和后置自增运算符将Date对象中天数加 1，必要时使年、月递增。Date 头文件（见图 11.12）指定了 Date 的 public 接口，包括下列函数：一个重载的流插入运算符（第11行）、一个默认的构造函数（第13行）、一个 setDate 函数（第14行）、一个重载的前置自增运算符（第15行）、一个重载的后置自增运算符（第16行）、一个重载的加法赋值运算符 +=（第17行）、一个对闰年进行测试的函数（第18行）和一个判断某天是否为当月最后一天的函数（第19行）。

```
1  // Fig. 11.12: Date.h
2  // Date class definition.
3  #ifndef DATE_H
4  #define DATE_H
5
6  #include <iostream>
7  using std::ostream;
8
9  class Date
10 {
11    friend ostream &operator<<( ostream &, const Date & );
12 public:
13    Date( int m = 1, int d = 1, int y = 1900 ); // default constructor
14    void setDate( int, int, int ); // set month, day, year
15    Date &operator++(); // prefix increment operator
16    Date operator++( int ); // postfix increment operator
17    const Date &operator+=( int ); // add days, modify object
18    bool leapYear( int ) const; // is date in a leap year?
19    bool endOfMonth( int ) const; // is date at the end of month?
```

图 11.12　具有重载的自增运算符的 Date 类的定义

```
20 private:
21    int month;
22    int day;
23    int year;
24
25    static const int days[]; // array of days per month
26    void helpIncrement(); // utility function for incrementing date
27 }; // end class Date
28
29 #endif
```

图 11.12 具有重载的自增运算符的Date类的定义（续）

```
 1 // Fig. 11.13: Date.cpp
 2 // Date class member-function definitions.
 3 #include <iostream>
 4 #include "Date.h"
 5
 6 // initialize static member at file scope; one classwide copy
 7 const int Date::days[] =
 8    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
 9
10 // Date constructor
11 Date::Date( int m, int d, int y )
12 {
13    setDate( m, d, y );
14 } // end Date constructor
15
16 // set month, day and year
17 void Date::setDate( int mm, int dd, int yy )
18 {
19    month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
20    year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
21
22    // test for a leap year
23    if ( month == 2 && leapYear( year ) )
24       day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
25    else
26       day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
27 } // end function setDate
28
29 // overloaded prefix increment operator
30 Date &Date::operator++()
31 {
32    helpIncrement(); // increment date
33    return *this; // reference return to create an lvalue
34 } // end function operator++
35
36 // overloaded postfix increment operator; note that the
37 // dummy integer parameter does not have a parameter name
38 Date Date::operator++( int )
39 {
40    Date temp = *this; // hold current state of object
41    helpIncrement();
42
43    // return unincremented, saved, temporary object
44    return temp; // value return; not a reference return
45 } // end function operator++
46
47 // add specified number of days to date
48 const Date &Date::operator+=( int additionalDays )
```

图 11.13 Date类成员函数和友元函数的定义

```
{
   for ( int i = 0; i < additionalDays; i++ )
      helpIncrement();

   return *this; // enables cascading
} // end function operator+=

// if the year is a leap year, return true; otherwise, return false
bool Date::leapYear( int testYear ) const
{
   if ( testYear % 400 == 0 ||
      ( testYear % 100 != 0 && testYear % 4 == 0 ) )
      return true; // a leap year
   else
      return false; // not a leap year
} // end function leapYear

// determine whether the day is the last day of the month
bool Date::endOfMonth( int testDay ) const
{
   if ( month == 2 && leapYear( year ) )
      return testDay == 29; // last day of Feb. in leap year
   else
      return testDay == days[ month ];
} // end function endOfMonth

// function to help increment the date
void Date::helpIncrement()
{
   // day is not end of month
   if ( !endOfMonth( day ) )
      day++; // increment day
   else
      if ( month < 12 ) // day is end of month and month < 12
      {
         month++; // increment month
         day = 1; // first day of new month
      } // end if
      else // last day of year
      {
         year++; // increment year
         month = 1; // first month of new year
         day = 1; // first day of new month
      } // end else
} // end function helpIncrement

// overloaded output operator
ostream &operator<<( ostream &output, const Date &d )
{
   static char *monthName[ 13 ] = { "", "January", "February",
      "March", "April", "May", "June", "July", "August",
      "September", "October", "November", "December" };
   output << monthName[ d.month ] << ' ' << d.day << ", " << d.year;
   return output; // enables cascading
} // end function operator<<
```

图 11.13 Date类成员函数和友元函数的定义（续）

图11.14中的main函数创建了3个Date对象（第11~13行）：默认初始化为1900年1月1日的d1、初始化为1992年12月27日的d2和初始化为一个无效日期的d3。Date构造函数（定义在图11.13中的第11~14行）调用setDate确定指定的月、日和年的有效性。无效的月份设置成1，无效的年份设置成1900，而无效的日则设置为1。

```
1  // Fig. 11.14: fig11_14.cpp
2  // Date class test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Date.h" // Date class definition
8
9  int main()
10 {
11    Date d1; // defaults to January 1, 1900
12    Date d2( 12, 27, 1992 ); // December 27, 1992
13    Date d3( 0, 99, 8045 ); // invalid date
14
15    cout << "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is " << d3;
16    cout << "\n\nd2 += 7 is " << ( d2 += 7 );
17
18    d3.setDate( 2, 28, 1992 );
19    cout << "\n\n  d3 is " << d3;
20    cout << "\n++d3 is " << ++d3 << " (leap year allows 29th)";
21
22    Date d4( 7, 13, 2002 );
23
24    cout << "\n\nTesting the prefix increment operator:\n"
25       << "  d4 is " << d4 << endl;
26    cout << "++d4 is " << ++d4 << endl;
27    cout << "  d4 is " << d4;
28
29    cout << "\n\nTesting the postfix increment operator:\n"
30       << "  d4 is " << d4 << endl;
31    cout << "d4++ is " << d4++ << endl;
32    cout << "  d4 is " << d4 << endl;
33    return 0;
34 } // end main
```

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992 (leap year allows 29th)

Testing the prefix increment operator:
  d4 is July 13, 2002
++d4 is July 14, 2002
  d4 is July 14, 2002

Testing the postfix increment operator:
  d4 is July 14, 2002
d4++ is July 14, 2002
  d4 is July 15, 2002
```

图 11.14　Date 类的测试程序

函数 main 的第 15~16 行使用重载的流插入运算符（定义在图 11.13 中的第 96~103 行）输出建立的每个 Date 对象。其中，第 16 行还使用重载的运算符 += 将 d2 增加了 7 天。第 18 行应用函数 setDate 把 d3 设置成一个闰年日期：1992 年 2 月 28 日。然后，第 20 行对 d3 进行前置自增，显示日期增加到 2 月 29 日。接着，第 22 行创建了一个 Date 对象 d4，用日期 2002 年 7 月 13 日对其进行初始化。然后第 26 行使用重载的前置自增运算符将 d4 增加 1 天。第 24~27 行输出应用前置

```
// Fig. 11.15: fig11_15.cpp
// Standard Library string class test program.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

int main()
{
   string s1( "happy" );
   string s2( " birthday" );
   string s3;

   // test overloaded equality and relational operators
   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
      << "\"; s3 is \"" << s3 << '\"'
      << "\n\nThe results of comparing s2 and s1:"
      << "\ns2 == s1 yields " << ( s2 == s1 ? "true" : "false" )
      << "\ns2 != s1 yields " << ( s2 != s1 ? "true" : "false" )
      << "\ns2 >  s1 yields " << ( s2 > s1 ? "true" : "false" )
      << "\ns2 <  s1 yields " << ( s2 < s1 ? "true" : "false" )
      << "\ns2 >= s1 yields " << ( s2 >= s1 ? "true" : "false" )
      << "\ns2 <= s1 yields " << ( s2 <= s1 ? "true" : "false" );

   // test string member-function empty
   cout << "\n\nTesting s3.empty():" << endl;

   if ( s3.empty() )
   {
      cout << "s3 is empty; assigning s1 to s3;" << endl;
      s3 = s1; // assign s1 to s3
      cout << "s3 is \"" << s3 << "\"";
   } // end if

   // test overloaded string concatenation operator
   cout << "\n\ns1 += s2 yields s1 = ";
   s1 += s2; // test overloaded concatenation
   cout << s1;

   // test overloaded string concatenation operator with C-style string
   cout << "\n\ns1 += \" to you\" yields" << endl;
   s1 += " to you";
   cout << "s1 = " << s1 << "\n\n";

   // test string member function substr
   cout << "The substring of s1 starting at location 0 for\n"
      << "14 characters, s1.substr(0, 14), is:\n"
      << s1.substr( 0, 14 ) << "\n\n";

   // test substr "to-end-of-string" option
   cout << "The substring of s1 starting at\n"
      << "location 15, s1.substr(15), is:\n"
      << s1.substr( 15 ) << endl;

   // test copy constructor
   string *s4Ptr = new string( s1 );
   cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";

   // test assignment (=) operator with self-assignment
   cout << "assigning *s4Ptr to *s4Ptr" << endl;
```

图 11.15　标准库 string 类

```
63      *s4Ptr = *s4Ptr;
64      cout << "*s4Ptr = " << *s4Ptr << endl;
65
66      // test destructor
67      delete s4Ptr;
68
69      // test using subscript operator to create lvalue
70      s1[ 0 ] = 'H';
71      s1[ 6 ] = 'B';
72      cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
73         << s1 << "\n\n";
74
75      // test subscript out of range with string member function "at"
76      cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
77      s1.at( 30 ) = 'd'; // ERROR: subscript out of range
78      return 0;
79   } // end main
```

```
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
s1 = happy birthday to you

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday

The substring of s1 starting at
location 15, s1.substr(15), is:
to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
*s4Ptr = happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1.at( 30 ) yields:

abnormal program termination
```

图 11.15　标准库 string 类（续）

第 12~14 行创建了 3 个 string 对象 s1、s2 和 s3，其中 s1 用字符串“happy”初始化，s2 用字符串“birthday”初始化，而 s3 利用默认的 string 构造函数创建了一个空的 string。第 17~18 行使用 cout 和运算符<<（string 类的设计者已经将它们重载以处理 string 对象）输出这 3 个对象。接着，第 19~25 行使用 string 类中重载的相等和关系运算符比较 s2 和 s1，并显示结果。

接收的参数是Array对象的const Array &。该函数输出其实参中元素的个数和内容。这里，这个Array对象的大小为7，因此输出7个0。

```
1  // Fig. 11.16: Fig11_16.cpp
2  // Driver for simple class Array.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Array.h"
8
9  void outputArray( const Array & ); // prototype
10
11 int main()
12 {
13    Array integers1( 7 ); // 7-element array
14    outputArray( integers1 ); // output Array integers1
15    outputArray( 3 ); // convert 3 to an Array and output Array's contents
16    return 0;
17 } // end main
18
19 // print Array contents
20 void outputArray( const Array &arrayToOutput )
21 {
22    cout << "The Array received has " << arrayToOutput.getSize()
23       << " elements. The contents are:\n" << arrayToOutput << endl;
24 } // end outputArray
```

```
The Array received has 7 elements. The contents are:
         0         0         0         0
         0         0         0

The Array received has 3 elements. The contents are:
         0         0         0
```

图 11.16　单参数的构造函数与隐式转换

第15行调用函数outputArray，以整数值3作为其实参。可是，这个程序并没有提供一个接收int参数的称为outputArray的函数。因此，编译器确定Array类是否提供了能把int转换成Array的转换构造函数。由于任何接收单参数的构造函数都可看做是转换构造函数，编译器就认为接收单个int参数的Array构造函数是一个转换构造函数，并用它将参数3转换成一个包含了3个元素的临时Array对象。然后，编译器把这个临时Array对象传递给函数outputArray，输出它的内容。这样一来，即使没有显式地提供一个接收int参数的outputArray函数，编译器也能够编译第15行。输出结果显示此3个元素Array对象的内容，它包含了3个0。

防止无意之中将单参数构造函数用做转换构造函数

C++提供了关键字explicit，用于禁止不应该允许的由转换构造函数完成的隐式转换。即，声明成explicit的构造函数不能在隐式转换中使用。图11.17在Array类中声明了一个explicit构造函数。Array.h的惟一更改是在第15行的单参数的构造函数声明前增加了关键字explicit，而包含了Array类成员函数定义的源代码文件不必做任何的变动。

图11.18给出了图11.16中程序的一个稍加修改的版本。当编译这个程序时，编译器会产生一条错误信息，指出在第15行中传递给outputArray的整数值无法转换成const Array &。编译器错误信息显示在输出窗口内。第16行示范了如何使用explicit构造函数创建一个包含3个元素的临时Array对象，并将它传递给outputArray函数。

```
1  // Fig. 11.17: Array.h
2  // Array class for storing arrays of integers.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 class Array
11 {
12    friend ostream &operator<<( ostream &, const Array & );
13    friend istream &operator>>( istream &, Array & );
14 public:
15    explicit Array( int = 10 ); // default constructor
16    Array( const Array & ); // copy constructor
17    ~Array(); // destructor
18    int getSize() const; // return size
19
20    const Array &operator=( const Array & ); // assignment operator
21    bool operator==( const Array & ) const; // equality operator
22
23    // inequality operator; returns opposite of == operator
24    bool operator!=( const Array &right ) const
25    {
26       return ! ( *this == right ); // invokes Array::operator==
27    } // end function operator!=
28
29    // subscript operator for non-const objects returns lvalue
30    int &operator[]( int );
31
32    // subscript operator for const objects returns rvalue
33    const int &operator[]( int ) const;
34 private:
35    int size; // pointer-based array size
36    int *ptr; // pointer to first element of pointer-based array
37 }; // end class Array
38
39 #endif
```

图 11.17　具有 explicit 构造函数的 Array 类的定义

```
1  // Fig. 11.18: Fig11_18.cpp
2  // Driver for simple class Array.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Array.h"
8
9  void outputArray( const Array & ); // prototype
10
11 int main()
12 {
13    Array integers1( 7 ); // 7-element array
14    outputArray( integers1 ); // output Array integers1
15    outputArray( 3 ); // convert 3 to an Array and output Array's contents
16    outputArray( Array( 3 ) ); // explicit single-argument constructor call
17    return 0;
18 } // end main
19
20 // print array contents
```

图 11.18　演示 explicit 构造函数

```
21  void outputArray( const Array &arrayToOutput )
22  {
23     cout << "The Array received has " << arrayToOutput.getSize()
24        << " elements. The contents are:\n" << arrayToOutput << endl;
25  } // end outputArray
```

```
c:\scpphtp5_examples\ch11\Fig11_17_18\Fig11_18.cpp(15) : error C2664:
'outputArray' : cannot convert parameter 1 from 'int' to 'const Array &'
        Reason: cannot convert from 'int' to 'const Array'
        Constructor for class 'Array' is declared 'explicit'
```

图 11.18　演示 explicit 构造函数（续）



常见的编程错误 11.10

试图调用 explicit 构造函数以进行隐式转换会导致编译错误。



常见的编程错误 11.11

关键字 explicit 只能用在单参数的构造函数前，如果用在数据成员或者其他成员函数前，则会导致编译错误。



错误预防技巧 11.3

对于那些不应被编译器用来执行隐式转换的单参数构造函数，请务必记住使用关键字 explicit。

11.15　本章小结

在本章中，学习了通过定义重载的运算符构建更加强大的类。重载运算符使程序员可以将类的对象当成 C++ 基本数据类型来看待。我们介绍了运算符重载的基本概念，以及 C++ 标准对重载运算符所附加的几个限制。让大家了解以成员函数或全局函数实现重载的运算符的原因，并分别讨论了以成员函数和全局函数方式重载一元和二元运算符的不同之处。就全局函数而言，分别示范了怎样用重载的流提取和重载的流插入运算符输入和输出类的对象。我们介绍了一种特殊的语法，必须使用它才能区分自增（++）运算符的前置和后置形式。我们还演示了标准 C++ 的类 string，这个强大的、可重用的类的创建大量使用了重载的运算符，它可以替代 C 风格的基于指针的字符串。在本章最后，学习了如何用关键字 explicit 防止编译器使用单参数构造函数执行隐式转换。在下一章中将通过介绍一种称为继承的软件重用形式，继续类的讨论。我们将看到，类常常共享公共的属性和行为。在这样的情况下，可以在一个公共的“基”类中定义这些属性和行为，然后将它们“继承”到新定义的类中。

摘要

- C++允许程序员重载大部分运算符，使运算符能符合所在的上下文环境——编译器基于上下文（尤其是操作数的类型）产生合适的代码。
- 许多 C++ 运算符可以重载以作用于用户自定义的类型。
- <<是 C++ 中内置的重载运算符的一个例子，它既用做流插入运算符，又用做按位左移运算符。同样，运算符>>也重载了，既用做流提取运算符，又用做按位右移运算符。这两个运算符都在 C++ 标准库中重载。
- C++语言本身就重载了加法运算符(+)和减法运算符(-)。视它们所在的运算环境是整数运算、浮点数运算还是指针运算，这两个运算符的执行会有所不同。
- 重载的运算符所执行的任务也可以由函数调用来完成。但是，对程序员来说，运算符的表示法往往更清晰，也更常见。

```
chessBoard( row, column )
```

这通过重载函数调用运算符来实现。

创建具有与图 11.6 和图 11.7 中 Array 类相似特性的类 DoubleSubscriptedArray。在构建时，该类应能创建任意行数和任意列数的数组。这个类应提供operator()执行双下标的操作。例如，在一个称为 a 的 3 × 5 的 DoubleSubscriptedArray 中，用户可以用 a(1, 3)访问第 1 行第 3 列处的元素。记住，operator()可以接收任意个数的参数（operator()的示例请见图 11.9 至图 11.10 中的 String 类）。双下标数组的基本表示应是一个单下标的整数数组，其元素个数为 rows*columns（行数乘以列数）。为访问数组的每个元素，函数 operator()必须执行正确的指针运算。应该有两个 operator()版本：一个返回 int &（从而使 DoubleSubscriptedArray 对象的元素可以用做左值），另一个返回 const int &（从而使 constDoubleSubscriptedArray 对象的元素只能用做右值）。该类还应提供下列运算符：==、!=、=、<<（以行和列格式输出数组）和>>（用来输入整个数组内容）。

11.12 重载下标运算符，使其返回集合中的最大元素、第二大元素、第三大元素，等等。

11.13 考虑图 11.19 至图 11.21 中给出的 Complex 类。该类可以对所谓的复数进行操作。复数的形式如下：realPart（实部）+ imaginaryPart（虚部）* i，其中 i 的值为 $\sqrt{-1}$。

a) 修改 Complex 类，使其通过重载的>>和<<运算符分别输入和输出复数（应该删除该类的 print 函数）。

b) 重载乘法运算符，使两个复数能够执行代数乘法。

c) 重载 == 和!= 运算符，支持复数之间的比较。

```
1  // Fig. 11.19: Complex.h
2  // Complex class definition.
3  #ifndef COMPLEX_H
4  #define COMPLEX_H
5
6  class Complex
7  {
8  public:
9     Complex( double = 0.0, double = 0.0 ); // constructor
10    Complex operator+( const Complex & ) const; // addition
11    Complex operator-( const Complex & ) const; // subtraction
12    void print() const; // output
13 private:
14    double real; // real part
15    double imaginary; // imaginary part
16 }; // end class Complex
17
18 #endif
```

图 11.19 Complex 类的定义

```
1  // Fig. 11.20: Complex.cpp
2  // Complex class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "Complex.h" // Complex class definition
7
8  // Constructor
9  Complex::Complex( double realPart, double imaginaryPart )
10    : real( realPart ),
11    imaginary( imaginaryPart )
12 {
```

图 11.20 Complex 类成员函数的定义

```
13       // empty body
14    } // end Complex constructor
15
16    // addition operator
17    Complex Complex::operator+( const Complex &operand2 ) const
18    {
19       return Complex( real + operand2.real,
20          imaginary + operand2.imaginary );
21    } // end function operator+
22
23    // subtraction operator
24    Complex Complex::operator-( const Complex &operand2 ) const
25    {
26       return Complex( real - operand2.real,
27          imaginary - operand2.imaginary );
28    } // end function operator-
29
30    // display a Complex object in the form: (a, b)
31    void Complex::print() const
32    {
33       cout << '(' << real << ", " << imaginary << ')';
34    } // end function print
```

图 11.20　Complex 类成员函数的定义（续）

```
1   // Fig. 11.21: fig11_21.cpp
2   // Complex class test program.
3   #include <iostream>
4   using std::cout;
5   using std::endl;
6
7   #include "Complex.h"
8
9   int main()
10  {
11     Complex x;
12     Complex y( 4.3, 8.2 );
13     Complex z( 3.3, 1.1 );
14
15     cout << "x: ";
16     x.print();
17     cout << "\ny: ";
18     y.print();
19     cout << "\nz: ";
20     z.print();
21
22     x = y + z;
23     cout << "\n\nx = y + z:" << endl;
24     x.print();
25     cout << " = ";
26     y.print();
27     cout << " + ";
28     z.print();
29
30     x = y - z;
31     cout << "\n\nx = y - z:" << endl;
32     x.print();
33     cout << " = ";
34     y.print();
35     cout << " - ";
36     z.print();
37     cout << endl;
```

图 11.21　复数

```
38      return 0;
39   } // end main
```

```
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z:
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)
```

图 11.21 复数（续）

11.14 32位整数的机器可以表示的整数范围大约是-20亿到20亿。这一范围的限制一般不会出现问题。不过在有些应用中，我们可能要使用更大范围内的整数。因此，需要创建强大的新数据类型，这也正是C++可以做到的。考虑图11.22至图11.24中的HugeInt类。仔细研究它，然后回答下列问题：

a) 准确描述这个类是如何操作的。

b) 这个类有哪些限制？

c) 重载乘法运算符*。

d) 重载除法运算符/。

e) 重载所有的关系和相等运算符。

注意：我们并没有给出HugeInteger类的赋值运算符或者拷贝构造函数，这是因为编译器提供的赋值运算符和拷贝构造函数就可以正确地复制整个数组数据成员。

```
1  // Fig. 11.22: Hugeint.h
2  // HugeInt class definition.
3  #ifndef HUGEINT_H
4  #define HUGEINT_H
5
6  #include <iostream>
7  using std::ostream;
8
9  class HugeInt
10 {
11    friend ostream &operator<<( ostream &, const HugeInt & );
12 public:
13    HugeInt( long = 0 ); // conversion/default constructor
14    HugeInt( const char * ); // conversion constructor
15
16    // addition operator; HugeInt + HugeInt
17    HugeInt operator+( const HugeInt & ) const;
18
19    // addition operator; HugeInt + int
20    HugeInt operator+( int ) const;
21
22    // addition operator;
23    // HugeInt + string that represents large integer value
24    HugeInt operator+( const char * ) const;
25 private:
26    short integer[ 30 ];
27 }; // end class HugetInt
28
29 #endif
```

图 11.22 HugeInt类的定义

```
1  // Fig. 11.23: Hugeint.cpp
2  // HugeInt member-function and friend-function definitions.
3  #include <cctype> // isdigit function prototype
4  #include <cstring> // strlen function prototype
5  #include "Hugeint.h" // HugeInt class definition
6
7  // default constructor; conversion constructor that converts
8  // a long integer into a HugeInt object
9  HugeInt::HugeInt( long value )
10 {
11    // initialize array to zero
12    for ( int i = 0; i <= 29; i++ )
13       integer[ i ] = 0;
14
15    // place digits of argument into array
16    for ( int j = 29; value != 0 && j >= 0; j-- )
17    {
18       integer[ j ] = value % 10;
19       value /= 10;
20    } // end for
21 } // end HugeInt default/conversion constructor
22
23 // conversion constructor that converts a character string
24 // representing a large integer into a HugeInt object
25 HugeInt::HugeInt( const char *string )
26 {
27    // initialize array to zero
28    for ( int i = 0; i <= 29; i++ )
29       integer[ i ] = 0;
30
31    // place digits of argument into array
32    int length = strlen( string );
33
34    for ( int j = 30 - length, k = 0; j <= 29; j++, k++ )
35
36       if ( isdigit( string[ k ] ) )
37          integer[ j ] = string[ k ] - '0';
38 } // end HugeInt conversion constructor
39
40 // addition operator; HugeInt + HugeInt
41 HugeInt HugeInt::operator+( const HugeInt &op2 ) const
42 {
43    HugeInt temp; // temporary result
44    int carry = 0;
45
46    for ( int i = 29; i >= 0; i-- )
47    {
48       temp.integer[ i ] =
49          integer[ i ] + op2.integer[ i ] + carry;
50
51       // determine whether to carry a 1
52       if ( temp.integer[ i ] > 9 )
53       {
54          temp.integer[ i ] %= 10;  // reduce to 0-9
55          carry = 1;
56       } // end if
57       else // no carry
58          carry = 0;
59    } // end for
60
61    return temp; // return copy of temporary object
62 } // end function operator+
```

图 11.23　HugeInt 类成员函数和友元函数的定义

```
63
64 // addition operator; HugeInt + int
65 HugeInt HugeInt::operator+( int op2 ) const
66 {
67    // convert op2 to a HugeInt, then invoke
68    // operator+ for two HugeInt objects
69    return *this + HugeInt( op2 );
70 } // end function operator+
71
72 // addition operator;
73 // HugeInt + string that represents large integer value
74 HugeInt HugeInt::operator+( const char *op2 ) const
75 {
76    // convert op2 to a HugeInt, then invoke
77    // operator+ for two HugeInt objects
78    return *this + HugeInt( op2 );
79 } // end operator+
80
81 // overloaded output operator
82 ostream& operator<<( ostream &output, const HugeInt &num )
83 {
84    int i;
85
86    for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= 29 ); i++ )
87       ; // skip leading zeros
88
89    if ( i == 30 )
90       output << 0;
91    else
92
93       for ( ; i <= 29; i++ )
94          output << num.integer[ i ];
95
96    return output;
97 } // end function operator<<
```

图 11.23 HugeInt类成员函数和友元函数的定义（续）

11.15 创建类RationalNumber（分数），使其具备下列能力：

a) 创建一个构造函数，它可以防止分数的分母为0。如果分数不是化简形式，它可以进行化简。而且，它还可以避免分母为负数。

b) 针对该类，重载加法、减法、乘法和除法运算符。

c) 针对该类，重载关系和相等运算符。

11.16 研究C语言的处理字符串的库函数，并将这些函数作为类String（见图11.9和图11.10）的一部分来实现。然后，使用它们进行文本操作。

11.17 开发类Polynomial（多项式）。在类内部，多项式由它的各个项组成的数组表示，每一项包含一个系数和一个指数。例如，项$2x^4$的系数为2，指数为4。开发一个完整的类，包含适当的构造函数、析构函数以及设置和获取函数。此外，Polynomial类还应提供下列重载运算符的能力：

a) 重载加法运算符（+），对两个Polynomial做加法。

b) 重载减法运算符（-），对两个Polynomial做减法。

c) 重载赋值运算符，把一个Polynomial赋给另一个。

d) 重载乘法运算符（*），对两个Polynomial做乘法。

e) 重载加法赋值运算符（+=）、减法赋值运算符（-=）和乘法赋值运算符（*=）。

11.18 在图 11.3 至图 11.5 的程序中，图 11.4 包含了注释“重载的流插入运算符；如果我们想以 cout << somePhoneNumber；调用重载的流插入运算符，那么它就不能是成员函数。”实际上，流插入运算符可以是 PhoneNumber 类的成员函数，只要愿意按 somePhoneNumber.operator<<(cout); 或者 somePhoneNumber << cout;的方式调用它。请重新编写图 11.5 的程序，其中重载的流插入运算符<<作为成员函数出现。另外，请在程序中测试上述两条语句，以证明它们可行。

```
// Fig. 11.24: fig11_24.cpp
// HugeInt test program.
#include <iostream>
using std::cout;
using std::endl;

#include "Hugeint.h"

int main()
{
   HugeInt n1( 7654321 );
   HugeInt n2( 7891234 );
   HugeInt n3( "99999999999999999999999999999" );
   HugeInt n4( "1" );
   HugeInt n5;

   cout << "n1 is " << n1 << "\nn2 is " << n2
      << "\nn3 is " << n3 << "\nn4 is " << n4
      << "\nn5 is " << n5 << "\n\n";

   n5 = n1 + n2;
   cout << n1 << " + " << n2 << " = " << n5 << "\n\n";

   cout << n3 << " + " << n4 << "\n= " << ( n3 + n4 ) << "\n\n";

   n5 = n1 + 9;
   cout << n1 << " + " << 9 << " = " << n5 << "\n\n";

   n5 = n2 + "10000";
   cout << n2 << " + " << "10000" << " = " << n5 << endl;
   return 0;
} // end main
```

```
n1 is 7654321
n2 is 7891234
n3 is 99999999999999999999999999999
n4 is 1
n5 is 0

7654321 + 7891234 = 15545555

99999999999999999999999999999 + 1
= 100000000000000000000000000000

7654321 + 9 = 7654330

7891234 + 10000 = 7901234
```

图 11.24　巨型整数

继承而来。在此，四边形类Quadrilateral是基类，矩形类Rectangle是派生类。一个矩形是一个特殊的四边形，但是如果断言四边形一定都是矩形，那是不对的。因为四边形中还有平行四边形或其他形状。图 12.1 列出了几种简单的基类和派生类的示例。

基类	派生类
学生	本科生、研究生
形状	圆、三角形、矩形、球体、立方体
贷款	汽车贷款、住房贷款、抵押贷款
雇员	教职员工、后勤人员
账户	支票账户、储蓄账户

图 12.1　继承示例

因为每个派生类的对象都“是”其基类的一个对象，并且一个基类可以有很多派生类，基类所能表示的对象集合一般比派生类所表示的对象集合要大。例如，基类Vehicle表示所有的交通工具，包括轿车、卡车、船、飞机、自行车，等等。相比之下，派生类Car只表示一个更小的更加特殊化的交通工具子集。

继承关系构成一种树状层次结构。基类和派生类间存在这种层次关系。虽然类可以独立存在，但是一旦投入到继承关系中，它们和其他的类就有了关联。也就是说，在继承关系中，类作为基类（为别的类提供成员）或是派生类（从别的类中继承成员），或二者兼有的身份而出现。

让我们设计一个简单继承层次，这个层次有五层（图 12.2 中由 UML 类图表示）。一个大学团体有数以千计的成员。

这些成员由Employee（雇员）、Student（学生）、Alumnus（校友）组成。Employee可能是Faculty（教职员工）或是Staff（后勤人员）。Faculty可能是Administrator（行政管理人员，例如校级领导和系主任）或是Teacher（教师）。有些Administrator，也可能授课。请注意，我们在设计类AdministratorTeacher（行政兼教师）时使用了多重继承，另外这个继承层次中还可能包括其他的类。例如，学生可以是研究生或本科生，本科生中又包括一年级、二年级、三年级和四年级学生。



图 12.2　大学社区成员 CommunityMembers 的继承层次结构

接下来看看图12.3中的Shape继承层次。这个层次由基类Shape开始。二维图形类TwoDimensionalShape和三维图形类ThreeDimensionalShape派生自基类Shape——Shape对象要么是TwoDimensionalShape对象，要么是ThreeDimensionalShape对象。该层次的第三层包含类 TwoDimensionalShape 和类ThreeDimensionalShape 的一些更具体的类型。和图 12.2 一样，我们也可以按箭头在这个类层次中由最底层至最高层的基类说出若干“is-a”关系。例如，Triangle是一个TwoDimensionalShape并且

是一个 Shape，一个 Sphere 是一个 ThreeDimensionalShape 并且是一个 Shape。请注意，这个层次中还包括许多其他的类，如 Rectangle、Ellipse 和 Trapezoid，这些都是 TwoDimensionalShape。

接下来看看图 12.3 中的 Shape 的继承层次。这个层次由基类 Shape 开始。二维图形类 TwoDimensionalShape 和三维图形类 ThreeDimensionalShape 派生自基类 Shape。该层次第三层中包括类 TwoDimensionalShape 和类 ThreeDimensionalShape 中更具体的类型。按图 12.2 中箭头指向关系，我们也可以指出本图中的“is-a”关系。例如，三角形 Triangle 是一个 TwoDimensionalShape 并且是一个 Shape。球体 Sphere 是一个 ThreeDimensionalShape 并且是一个 Shape。这个层次中还包括其他的类，如 Rectanyle、Ellipse 和 Trapezoid，这些都是 TwoDimensionalShape。

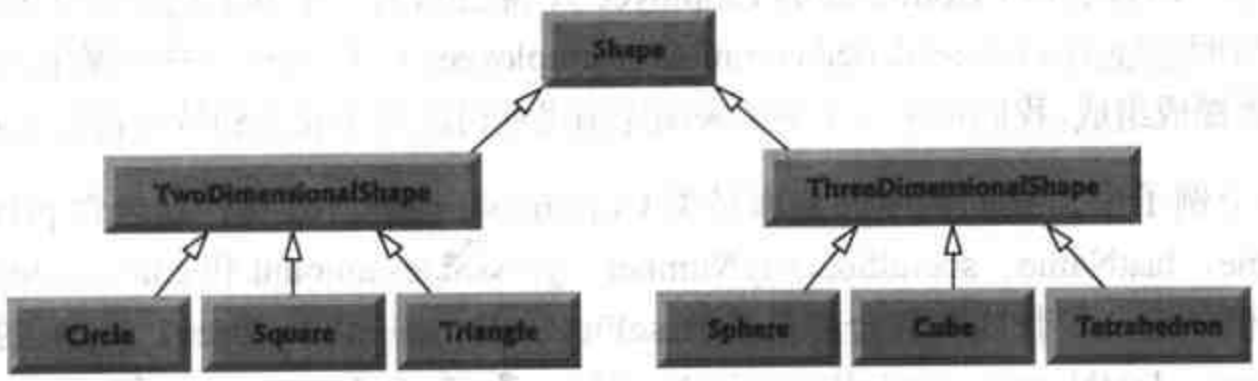


图 12.3　Shape 继承层次

为了指定类 TwoDimensionalShape 是由类 Shape 派生（继承）而来的（见图 12.3），类 TwoDimensionalShape 在 C++ 中可以以如下语句定义：

```
class TwoDimensionalShape : public Shape
```

这是最常使用的继承形式:public 继承的一个例子。我们还将在 12.6 节讨论 private 继承和 protected 继承。在各种形式的继承中，基类的 private 成员都不能被其派生类直接访问，但 private 基类成员仍得到了继承（即它们仍被视为派生类的一部分）。在 public 继承中，基类的所有其他成员在成为派生类的成员时仍保持了其原始的成员访问权限（例如，基类的 public 成员在派生类中仍是 pulic 成员，基类的 protected 成员在派生类中仍是 protected 成员）。通过这些继承于基类的成员，派生类能够操纵基类中的 private 成员（但需要基类中这些要继承的成员提供如此的功能）。值得注意的是，friend 函数是不能被继承的。

继承并不适合所有的类关系。在第 10 章中，我们讨论过“has-a”关系，这种关系中类的成员可以是其他类的对象。例如，给定类 Employee、BirthDate 和 TelephoneNumber，那么无论说一个 Employee 是一个 BirthDate 或者说一个 Employee 是一个 TelephoneNumber 都不正确。我们只能说一个 Employee 有一个 BirthDate 或者说一个 Employee 有一个 TelephoneNumber。

用相似的方法处理基类和派生类是可能的，它们的共性都体现在基类的成员中。由共同基类派生出的所有类的对象均可视为这个基类的对象，也就是说，这个对象与基类具有“is-a”关系。在第 13 章中，我们将考虑利用这种关系的许多例子。

12.3　protected 成员

第 3 章介绍了 public 和 private 访问说明符。基类的 public 成员在它自己的体内是可访问的，而且在程序中凡是出现该基类或其派生类的对象句柄（即基类或派生类对象的名字、引用或指针）处也是可访问的。但是基类的 private 成员只能在此基类体内以及友元函数中访问。本节将介绍另一种成员访问说明符：protected。

protected 访问提供了一种介于 public 和 private 访问之间的中间保护层次。基类的 protected 成员可以在基类的体内被基类的成员和友元访问，也可以由基类所派生出的任何类的成员和友元访问。

前几个例子中，类CommissionEmployee构造函数的定义特意没有使用成员初始化器语法，这样可以演示private和protected成员访问说明符是如何影响派生类中的成员访问的。图12.5第13~15行中，在构造函数的函数体中对数据成员 firstName,lastName 和 socialSecurityNumber进行赋值。本节后面，将在构造函数中使用成员初始化器列表。

请注意，这里我们在将构造函数的参数值赋给相应的数据成员之前，没有确认这些值的有效性。当然，我们可以确认名和姓的有效性，也许可以保证一个合理的长度。类似地，可以确认社会安全号码的有效性，以保证它包含有或没有破折号的9位数表示（例如，123-45-6789或123456789）。

```
1  // Fig. 12.4: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate (percentage)
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

图 12.4　类 CommissionEmployee 的头文件

```
1  // Fig. 12.5: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  // constructor
9  CommissionEmployee::CommissionEmployee(
10    const string &first, const string &last, const string &ssn,
11    double sales, double rate )
```

图 12.5　表示抽取一定比例销售提成的雇员类 CommissionEmployee 的实现文件

```
12 {
13    firstName = first; // should validate
14    lastName = last;   // should validate
15    socialSecurityNumber = ssn; // should validate
16    setGrossSales( sales ); // validate and store gross sales
17    setCommissionRate( rate ); // validate and store commission rate
18 } // end CommissionEmployee constructor
19
20 // set first name
21 void CommissionEmployee::setFirstName( const string &first )
22 {
23    firstName = first; // should validate
24 } // end function setFirstName
25
26 // return first name
27 string CommissionEmployee::getFirstName() const
28 {
29    return firstName;
30 } // end function getFirstName
31
32 // set last name
33 void CommissionEmployee::setLastName( const string &last )
34 {
35    lastName = last; // should validate
36 } // end function setLastName
37
38 // return last name
39 string CommissionEmployee::getLastName() const
40 {
41    return lastName;
42 } // end function getLastName
43
44 // set social security number
45 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
46 {
47    socialSecurityNumber = ssn; // should validate
48 } // end function setSocialSecurityNumber
49
50 // return social security number
51 string CommissionEmployee::getSocialSecurityNumber() const
52 {
53    return socialSecurityNumber;
54 } // end function getSocialSecurityNumber
55
56 // set gross sales amount
57 void CommissionEmployee::setGrossSales( double sales )
58 {
59    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
60 } // end function setGrossSales
61
62 // return gross sales amount
63 double CommissionEmployee::getGrossSales() const
64 {
65    return grossSales;
66 } // end function getGrossSales
67
68 // set commission rate
69 void CommissionEmployee::setCommissionRate( double rate )
70 {
71    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
72 } // end function setCommissionRate
73
74 // return commission rate
75 double CommissionEmployee::getCommissionRate() const
76 {
77    return commissionRate;
```

图 12.5　表示抽取一定比例销售提成的雇员类 CommissionEmployee 的实现文件（续）

```
78  } // end function getCommissionRate
79
80  // calculate earnings
81  double CommissionEmployee::earnings() const
82  {
83     return commissionRate * grossSales;
84  } // end function earnings
85
86  // print CommissionEmployee object
87  void CommissionEmployee::print() const
88  {
89     cout << "commission employee: " << firstName << ' ' << lastName
90        << "\nsocial security number: " << socialSecurityNumber
91        << "\ngross sales: " << grossSales
92        << "\ncommission rate: " << commissionRate;
93  } // end function print
```

图 12.5　表示抽取一定比例销售提成的雇员类 CommissionEmployee 的实现文件（续）

成员函数 earnings（第 81~84 行）计算 CommissionEmployee 对象的收入。第 83 行计算 commissionRate 和 grossSales 的乘积并返回结果。成员函数 print（第 87~93 行）显示 CommissionEmployee 对象数据成员的值。

图 12.6 测试了类 CommissionEmployee。程序的第 16~17 行实例化类 CommissionEmployee 的一个对象 employee，并调用 CommissionEmployee 构造函数初始化这个对象，即用 "Sue" 初始化名，"Jones" 初始化姓，"222-22-2222" 初始化社会安全号码，10000 初始化销售总额，0.06 初始化提成比例。第 23~29 行用 employee 的各个获取函数显示它的数据成员的值。第 31~32 行调用了该对象的 setGrossSales 和 setCommissionRate 成员函数分别改变数据成员 grossSales 和 commissionRate 的值。然后第 36 行调用 employee 的成员函数 print 输出更新后此 CommissionEmployee 对象中的信息。最后，第 39 行用成员函数 earnings 计算数据成员 grossSales 和 commissionRate 已更新值的乘积，从而得到这个 CommissionEmployee 对象的收入值，并显示出来。

```
1  // Fig. 12.6: fig12_06.cpp
2  // Testing class CommissionEmployee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 #include "CommissionEmployee.h" // CommissionEmployee class definition
12
13 int main()
14 {
15    // instantiate a CommissionEmployee object
16    CommissionEmployee employee(
17       "Sue", "Jones", "222-22-2222", 10000, .06 );
18
19    // set floating-point output formatting
20    cout << fixed << setprecision( 2 );
21
22    // get commission employee data
23    cout << "Employee information obtained by get functions: \n"
24       << "\nFirst name is " << employee.getFirstName()
25       << "\nLast name is " << employee.getLastName()
26       << "\nSocial security number is "
27       << employee.getSocialSecurityNumber()
28       << "\nGross sales is " << employee.getGrossSales()
```

图 12.6　类 CommissionEmployee 的测试程序和输出结果

```
29        << "\nCommission rate is " << employee.getCommissionRate() << endl;
30
31     employee.setGrossSales( 8000 ); // set gross sales
32     employee.setCommissionRate( .1 ); // set commission rate
33
34     cout << "\nUpdated employee information output by print function: \n"
35        << endl;
36     employee.print(); // display the new employee information
37
38     // display the employee's earnings
39     cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
40
41     return 0;
42  } // end main
```

```
Employee information obtained by get functions:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information output by print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 8000.00
commission rate: 0.10

Employee's earnings: $800.00
```

图 12.6　类 CommissionEmployee 的测试程序和输出结果（续）

12.4.2　不使用继承创建类 BasePlusCommissionEmployee

现在进入我们对继承讨论的第二部分：创建并测试一个全新且独立的类 BasePlusCommissionEmployee（见图 12.7 和图 12.8），它包括数据成员 firstName、lastName、socialSecurityNumber、grossSales、commissionRate 和 baseSalary。

```
1  // Fig. 12.7: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class definition represents an employee
3  // that receives a base salary in addition to commission.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 class BasePlusCommissionEmployee
11 {
12 public:
13    BasePlusCommissionEmployee( const string &, const string &,
14       const string &, double = 0.0, double = 0.0, double = 0.0 );
15
16    void setFirstName( const string & ); // set first name
17    string getFirstName() const; // return first name
18
19    void setLastName( const string & ); // set last name
20    string getLastName() const; // return last name
21
```

图 12.7　类 BasePlusCommissionEmployee 的头文件

```
22     void setSocialSecurityNumber( const string & ); // set SSN
23     string getSocialSecurityNumber() const; // return SSN
24
25     void setGrossSales( double ); // set gross sales amount
26     double getGrossSales() const; // return gross sales amount
27
28     void setCommissionRate( double ); // set commission rate
29     double getCommissionRate() const; // return commission rate
30
31     void setBaseSalary( double ); // set base salary
32     double getBaseSalary() const; // return base salary
33
34     double earnings() const; // calculate earnings
35     void print() const; // print BasePlusCommissionEmployee object
36  private:
37     string firstName;
38     string lastName;
39     string socialSecurityNumber;
40     double grossSales; // gross weekly sales
41     double commissionRate; // commission percentage
42     double baseSalary; // base salary
43  }; // end class BasePlusCommissionEmployee
44
45  #endif
```

图 12.7　类 BasePlusCommissionEmployee 的头文件（续）

```
1   // Fig. 12.8: BasePlusCommissionEmployee.cpp
2   // Class BasePlusCommissionEmployee member-function definitions.
3   #include <iostream>
4   using std::cout;
5
6   // BasePlusCommissionEmployee class definition
7   #include "BasePlusCommissionEmployee.h"
8
9   // constructor
10  BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate, double salary )
13  {
14     firstName = first; // should validate
15     lastName = last; // should validate
16     socialSecurityNumber = ssn; // should validate
17     setGrossSales( sales ); // validate and store gross sales
18     setCommissionRate( rate ); // validate and store commission rate
19     setBaseSalary( salary ); // validate and store base salary
20  } // end BasePlusCommissionEmployee constructor
21
22  // set first name
23  void BasePlusCommissionEmployee::setFirstName( const string &first )
24  {
25     firstName = first; // should validate
26  } // end function setFirstName
27
28  // return first name
29  string BasePlusCommissionEmployee::getFirstName() const
30  {
31     return firstName;
32  } // end function getFirstName
33
34  // set last name
35  void BasePlusCommissionEmployee::setLastName( const string &last )
```

图 12.8　以底薪加提成为薪水的雇员类 BasePlusCommissionEmployee

```
36 {
37    lastName = last; // should validate
38 } // end function setLastName
39
40 // return last name
41 string BasePlusCommissionEmployee::getLastName() const
42 {
43    return lastName;
44 } // end function getLastName
45
46 // set social security number
47 void BasePlusCommissionEmployee::setSocialSecurityNumber(
48    const string &ssn )
49 {
50    socialSecurityNumber = ssn; // should validate
51 } // end function setSocialSecurityNumber
52
53 // return social security number
54 string BasePlusCommissionEmployee::getSocialSecurityNumber() const
55 {
56    return socialSecurityNumber;
57 } // end function getSocialSecurityNumber
58
59 // set gross sales amount
60 void BasePlusCommissionEmployee::setGrossSales( double sales )
61 {
62    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
63 } // end function setGrossSales
64
65 // return gross sales amount
66 double BasePlusCommissionEmployee::getGrossSales() const
67 {
68    return grossSales;
69 } // end function getGrossSales
70
71 // set commission rate
72 void BasePlusCommissionEmployee::setCommissionRate( double rate )
73 {
74    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
75 } // end function setCommissionRate
76
77 // return commission rate
78 double BasePlusCommissionEmployee::getCommissionRate() const
79 {
80    return commissionRate;
81 } // end function getCommissionRate
82
83 // set base salary
84 void BasePlusCommissionEmployee::setBaseSalary( double salary )
85 {
86    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
87 } // end function setBaseSalary
88
89 // return base salary
90 double BasePlusCommissionEmployee::getBaseSalary() const
91 {
92    return baseSalary;
93 } // end function getBaseSalary
94
95 // calculate earnings
96 double BasePlusCommissionEmployee::earnings() const
97 {
```

图 12.8　以底薪加提成为薪水的雇员类 BasePlusCommissionEmployee（续）

```
98       return baseSalary + ( commissionRate * grossSales );
99    } // end function earnings
100
101   // print BasePlusCommissionEmployee object
102   void BasePlusCommissionEmployee::print() const
103   {
104      cout << "base-salaried commission employee: " << firstName << ' '
105         << lastName << "\nsocial security number: " << socialSecurityNumber
106         << "\ngross sales: " << grossSales
107         << "\ncommission rate: " << commissionRate
108         << "\nbase salary: " << baseSalary;
109   } // end function print
```

图 12.8　以底薪加提成为薪水的雇员类 BasePlusCommissionEmployee（续）

定义类 BasePlusCommissionEmployee

类BasePlusCommissionEmployee的头文件（见图12.7）描述了类BasePlusCommissionEmployee提供的public服务，包括类BasePlusCommissionEmployee的构造函数（第13~14行）、成员函数earnings（第34行）和print（第35行）。第16~32行对于类的private数据成员（在第37~42行声明）firstName、lastName、socialSecurityNumber、grossSales、commissionRate和baseSalary声明了public的设置和获取函数。这些变量和成员函数封装了带底薪佣金雇员的所有必要的特征。请注意这个类与类CommissionEmployee的相似之处（见图12.4和图12.5），但是本例中，我们还没有利用到这种相似性。

类BasePlusCommissionEmployee的成员函数earnings（定义于图12.8的第96~99行）计算带底薪佣金雇员的收入。第98行返回雇员的提成比例commission rate与销售总额gross sales的乘积再加上底薪base salary而得到的值。

测试类 BasePlusCommissionEmployee

图 12.9 对类 BasePlusCommissionEmployee 进行了测试。第 17-18 行实例化类 BasePlusCommissionEmployee 的对象 employee，分别以 "Bob"、"Lewis"、"333-33-3333"、5000,0.04 和 300 传递给构造函数作为firstName、lastName、socialSecurityNumber、grossSales、commissionRate和baseSalary的值。第24~31行利用类BasePlusCommissionEmployee的各个获取函数取得该对象的数据成员值用于输出。第33行调用对象的setBaseSalary成员函数更改底薪的值。因为雇员的底薪不可能为负值，所以成员函数setBaseSalary（图12.8中第84~87行）将保证数据成员baseSalary不被赋为负值。图12.9中第37行调用对象的成员函数print，输出更新后此BasePlusCommissionEmployee对象的信息，第40行调用成员函数earnings，显示该BasePlusCommissionEmployee对象的收入。

考察类 BasePlusCommissionEmployee 和类 CommissionEmployee 之间的相似性

通过对比，可以看到类BasePlusCommissionEmployee（见图12.7和图12.8）中的大量代码如果不相同，则都和类CommissionEmployee（见图12.4和图12.5）中的相似。例如，类BasePlusCommissionEmployee中的private数据成员firstName和lastName，成员函数setFirstName、getFirstName、setLastName、getLastName都与类CommissionEmployee中的一模一样。还有类CommissionEmployee和类BasePlusCommissionEmployee都包括private数据成员socialSecurityNumber、commissionRate和grossSales,以及操作这些数据的设置和获取函数。除了类BasePlusCommissionEmployee的构造函数还需设定baseSalary之外，类BasePlusCommissionEmployee的构造函数和类CommissionEmployee的几乎是相同的。类BasePlusCommissionEmployee添加了private数据成员baseSalary，成员函数setBaseSalary和getBaseSalary。类BasePlusCommissionEmployee的成员函数print和类CommissionEmployee的也几乎相同，除了类BasePlusCommissionEmployee中的print还要输出数据成员baseSalary的值之外。

```
// Fig. 12.9: fig12_09.cpp
// Testing class BasePlusCommissionEmployee.
#include <iostream>
using std::cout;
using std::endl;
using std::fixed;

#include <iomanip>
using std::setprecision;

// BasePlusCommissionEmployee class definition
#include "BasePlusCommissionEmployee.h"

int main()
{
   // instantiate BasePlusCommissionEmployee object
   BasePlusCommissionEmployee
      employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

   // get commission employee data
   cout << "Employee information obtained by get functions: \n"
      << "\nFirst name is " << employee.getFirstName()
      << "\nLast name is " << employee.getLastName()
      << "\nSocial security number is "
      << employee.getSocialSecurityNumber()
      << "\nGross sales is " << employee.getGrossSales()
      << "\nCommission rate is " << employee.getCommissionRate()
      << "\nBase salary is " << employee.getBaseSalary() << endl;

   employee.setBaseSalary( 1000 ); // set base salary

   cout << "\nUpdated employee information output by print function: \n"
      << endl;
   employee.print(); // display the new employee information

   // display the employee's earnings
   cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;

   return 0;
} // end main
```

```
Employee information obtained by get functions:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information output by print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00
```

图 12.9　类 BasePlusCommissionEmployee 的测试程序

我们逐字逐句地复制 CommissionEmployee 的代码并把它粘贴到类 BasePlusCommissionEmployee 中，之后修改类 BasePlusCommissionEmployee，使之包括底薪和操作该底薪的成员函数。这种“复制–粘贴”的方法常常容易出错且耗费时间。更糟的是，如果这种方法的结果遍布于整个系

统，即相同代码的多个物理拷贝存在于系统中，则对于代码维护人员而言这简直是场恶梦。那么，有没有办法可以“吸收”类的数据成员和成员函数，而不用复制这些代码，使它们成为其他类的一部分呢？在接下来的几个例子中，我们将用继承机制解决这个问题。



软件工程知识 12.3
从一个类向另一个类复制粘贴代码，可能造成错误在多个源代码文件中扩散。当我们想要一个类“吸收”别的类的数据成员和成员函数时，使用继承而不是“复制粘贴”手段，从而避免代码（和可能错误）的复制。



软件工程知识 12.4
使用继承时，类层次中所有类共同的数据成员和成员函数在基类中声明。当需要对这些共同特征进行修改时，程序员只需在基类中进行修改，于是派生类也就继承了相应修改。如果不采用继承机制，则需要对所有包含代码拷贝的源代码文件进行修改。

12.4.3　创建 CommissionEmployee —— BasePlusCommissionEmployee 继承层次

现在我们创建并测试一个新版本的类 BasePlusCommissionEmployee（见图 12.10 和图 12.11），它由类 CommissionEmployee（见图 12.4 和图 12.5）派生。此例中类 BasePlusCommissionEmployee 的对象同时是一个 CommissionEmployee 对象（因为继承了类 CommissionEmployee 中的所有性能），但类 BasePlusCommissionEmployee 还具有数据成员 baseSalary（图 12.10 中第 24 行）。此类定义中第 12 行的冒号（:）表示这是一个继承，关键词 public 指明该继承的类型。作为派生类，类 BasePlusCommissionEmployee 继承了类 CommissionEmployee 中除构造函数的所有成员，每个类都提供特定于自己的构造函数。注意：析构函数同样不能继承。因此，类 BasePlusCommissionEmployee 提供的 public 服务包括自己的构造函数和从类 CommissionEmployee 继承而来的 public 成员函数（第 15~16 行），虽然我们在类 BasePlusCommissionEmployee 中看不到这些继承来的成员函数的代码，但它们的确是派生类 BasePlusCommissionEmployee 的一部分。该派生类提供的 public 服务还包括成员函数 setBaseSalary、getBaseSalary、earnings 和 print（第 18~22 行）。

```
1  // Fig. 12.10: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    double earnings() const; // calculate earnings
22    void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

图 12.10　指明了与 CommissionEmployee 类继承关系的类 BasePlusCommissionEmployee 的定义

图 12.11 实现了类 BasePlusCommissionEmployee 的成员函数。构造函数（第 10~17 行）引入了基类初始化器语法（第 14 行），使用成员初始化器将参数传递给基类（CommissionEmployee）的构造函数。C++要求派生类构造函数调用其基类的构造函数，来初始化继承到派生类的基类的数据成员。为此，第 14 行通过名字调用类 CommissionEmployee 的构造函数，将当前构造函数的参数 first、last、ssn、sales 和 rate 作为实参传递给它，以初始化基类的数据成员 firstName、lastName、socialSecurityNumber、grossSales 和 commissionRate。如果类 BasePlusCommissionEmployee 的构造函数没有显式地调用基类 CommissionEmployee 的构造函数，C++ 将尝试隐式地调用类 CommissionEmployee 的默认构造函数，但是该类却没有这样的默认构造函数，所以编译器发布一则错误信息。回忆一下在第 3 章曾讲过，编译器为没有显示提供构造函数的类提供一个没有参数的默认构造函数。然而，类 CommissionEmployee 确实有显式的构造函数，因此编译器不再提供默认构造函数。当试图隐式地调用 CommissionEmployee 的默认构造函数时，就会产生编译错误。

```
1  // Fig. 12.11: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string &first, const string &last, const string &ssn,
12    double sales, double rate, double salary )
13    // explicitly call base-class constructor
14    : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16    setBaseSalary( salary ); // validate and store base salary
17 } // end BasePlusCommissionEmployee constructor
18
19 // set base salary
20 void BasePlusCommissionEmployee::setBaseSalary( double salary )
21 {
22    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // end function setBaseSalary
24
25 // return base salary
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28    return baseSalary;
29 } // end function getBaseSalary
30
31 // calculate earnings
32 double BasePlusCommissionEmployee::earnings() const
33 {
34    // derived class cannot access the base class's private data
35    return baseSalary + ( commissionRate * grossSales );
36 } // end function earnings
37
38 // print BasePlusCommissionEmployee object
39 void BasePlusCommissionEmployee::print() const
40 {
41    // derived class cannot access the base class's private data
42    cout << "base-salaried commission employee: " << firstName << ' '
43       << lastName << "\nsocial security number: " << socialSecurityNumber
44       << "\ngross sales: " << grossSales
45       << "\ncommission rate: " << commissionRate
46       << "\nbase salary: " << baseSalary;
```

图 12.11 类 BasePlusCommissionEmployee 的实现文件：派生类不能访问基类的 private 数据

```
47   } // end function print
```

```
C:\scpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(35) :
   error C2248: 'CommissionEmployee::commissionRate' :
   cannot access private member declared in class 'CommissionEmployee'
      C:\scpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(37) :
         see declaration of 'CommissionEmployee::commissionRate'
      C:\scpphtp5e_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'

C:\scpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(35) :
   error C2248: 'CommissionEmployee::grossSales' :
   cannot access private member declared in class 'CommissionEmployee'
      C:\scpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(36) :
         see declaration of 'CommissionEmployee::grossSales'
      C:\scpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'

C:\scpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(42) :
   error C2248: 'CommissionEmployee::firstName' :
   cannot access private member declared in class 'CommissionEmployee'
      C:\scpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(33) :
         see declaration of 'CommissionEmployee::firstName'
      C:\scpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'

C:\scpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(43) :
   error C2248: 'CommissionEmployee::lastName' :
   cannot access private member declared in class 'CommissionEmployee'
      C:\scpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(34) :
         see declaration of 'CommissionEmployee::lastName'
      C:\scpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'

C:\scpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(43) :
   error C2248: 'CommissionEmployee::socialSecurity-Number' :
   cannot access private member declared in class 'CommissionEmployee'
      C:\scpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(35) :
         see declaration of 'CommissionEmployee::socialSecurityNumber'
      C:\scpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'

C:\scpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(44) :
   error C2248: 'CommissionEmployee::grossSales' :
   cannot access private member declared in class 'CommissionEmployee'
      C:\scpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(36) :
         see declaration of 'CommissionEmployee::grossSales'
      C:\scpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'

C:\scpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(45) :
   error C2248: 'CommissionEmployee::commissionRate' :
   cannot access private member declared in class 'CommissionEmployee'
      C:\scpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(37) :
         see declaration of 'CommissionEmployee::commissionRate'
      C:\scpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
```

图 12.11　类 BasePlusCommissionEmployee 的实现文件：派生类不能访问基类的 private 数据（续）



常见的编程错误 12.1

派生类构造函数调用其基类构造函数时，如果传递给基类构造函数的参数，它的个数和类型与基类构造函数的相应定义不符，将导致编译错误。



性能提示 12.1

在派生类的构造函数中，采用成员初始化器列表显式地初始化成员对象和调用基类的构造函数，可以防止重复初始化，即调用了默认构造函数之后，又在派生类构造函数中再次修改数据成员。

些成员声明为protected。正如12.3节中所介绍的，基类的protected成员既可以被基类的成员和友元访问，又可以被由基类派生的任何类的成员和友元访问。



良好的编程习惯 12.1

最好是首先声明 public 成员，然后是 protected 成员，最后是 private 成员。

定义使用 protected 数据的基类 CommissionEmployee

现在，类 CommissionEmployee（见图 12.12 和图 12.13）把数据成员 firstName、lastName、socialSecurityNumber、grossSales 和 commissionRate 声明为 protected 而不是 private（见图 12.12 中的第 33~37 行）。图 12.13 的成员函数的实现和图 12.5 中的相同。

```
1  // Fig. 12.12: CommissionEmployee.h
2  // CommissionEmployee class definition with protected data.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 protected:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

图 12.12　声明了允许派生类访问的 protected 数据的类 CommissionEmployee 的定义

```
1  // Fig. 12.13: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
```

图 12.13　具有 protected 数据的类 CommissionEmployee

```
 7
 8  // constructor
 9  CommissionEmployee::CommissionEmployee(
10     const string &first, const string &last, const string &ssn,
11     double sales, double rate )
12  {
13     firstName = first; // should validate
14     lastName = last; // should validate
15     socialSecurityNumber = ssn; // should validate
16     setGrossSales( sales ); // validate and store gross sales
17     setCommissionRate( rate ); // validate and store commission rate
18  } // end CommissionEmployee constructor
19
20  // set first name
21  void CommissionEmployee::setFirstName( const string &first )
22  {
23     firstName = first; // should validate
24  } // end function setFirstName
25
26  // return first name
27  string CommissionEmployee::getFirstName() const
28  {
29     return firstName;
30  } // end function getFirstName
31
32  // set last name
33  void CommissionEmployee::setLastName( const string &last )
34  {
35     lastName = last; // should validate
36  } // end function setLastName
37
38  // return last name
39  string CommissionEmployee::getLastName() const
40  {
41     return lastName;
42  } // end function getLastName
43
44  // set social security number
45  void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
46  {
47     socialSecurityNumber = ssn; // should validate
48  } // end function setSocialSecurityNumber
49
50  // return social security number
51  string CommissionEmployee::getSocialSecurityNumber() const
52  {
53     return socialSecurityNumber;
54  } // end function getSocialSecurityNumber
55
56  // set gross sales amount
57  void CommissionEmployee::setGrossSales( double sales )
58  {
59     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
60  } // end function setGrossSales
61
62  // return gross sales amount
63  double CommissionEmployee::getGrossSales() const
64  {
65     return grossSales;
66  } // end function getGrossSales
67
68  // set commission rate
69  void CommissionEmployee::setCommissionRate( double rate )
70  {
71     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
```

图 12.13　具有 protected 数据的类 CommissionEmployee（续）

```
72  } // end function setCommissionRate
73
74  // return commission rate
75  double CommissionEmployee::getCommissionRate() const
76  {
77     return commissionRate;
78  } // end function getCommissionRate
79
80  // calculate earnings
81  double CommissionEmployee::earnings() const
82  {
83     return commissionRate * grossSales;
84  } // end function earnings
85
86  // print CommissionEmployee object
87  void CommissionEmployee::print() const
88  {
89     cout << "commission employee: " << firstName << ' ' << lastName
90        << "\nsocial security number: " << socialSecurityNumber
91        << "\ngross sales: " << grossSales
92        << "\ncommission rate: " << commissionRate;
93  } // end function print
```

图 12.13　具有 protected 数据的类 CommissionEmployee（续）

修改派生类 BasePlusCommissionEmployee

下面，我们对类 BasePlusCommissionEmployee（见图 12.14 和图 12.15）进行修改，使之继承图 12.12 和图 12.13 版本的类 CommissionEmployee。因为类 BasePlusCommissionEmployee 继承了该版的类 CommissionEmployee，类 BasePlusCommissionEmployee 的对象可以直接访问继承于类 CommissionEmployee 中声明为 protected 的数据成员（也就是数据成员 firstName、lastName、socialSecurityNumber、grossSales 和 commissionRate）。这样，编译器在编译定义在图 12.15 中的类 BasePlusCommissionEmployee 的成员函数 earnings（第 32~36 行）和 print（第 39~47 行）时就不会产生错误。这表明派生类获得了访问protected基类数据成员的特权。派生类对象也可以访问该派生类任何间接基类中的 protected 成员。

```
1   // Fig. 12.14: BasePlusCommissionEmployee.h
2   // BasePlusCommissionEmployee class derived from class
3   // CommissionEmployee.
4   #ifndef BASEPLUS_H
5   #define BASEPLUS_H
6
7   #include <string> // C++ standard string class
8   using std::string;
9
10  #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12  class BasePlusCommissionEmployee : public CommissionEmployee
13  {
14  public:
15     BasePlusCommissionEmployee( const string &, const string &,
16        const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18     void setBaseSalary( double ); // set base salary
19     double getBaseSalary() const; // return base salary
20
21     double earnings() const; // calculate earnings
22     void print() const; // print BasePlusCommissionEmployee object
23  private:
24     double baseSalary; // base salary
25  }; // end class BasePlusCommissionEmployee
26
27  #endif
```

图 12.14　类 BasePlusCommissionEmployee 的头文件

```
1  // Fig. 12.15: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string &first, const string &last, const string &ssn,
12    double sales, double rate, double salary )
13    // explicitly call base-class constructor
14    : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16    setBaseSalary( salary ); // validate and store base salary
17 } // end BasePlusCommissionEmployee constructor
18
19 // set base salary
20 void BasePlusCommissionEmployee::setBaseSalary( double salary )
21 {
22    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // end function setBaseSalary
24
25 // return base salary
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28    return baseSalary;
29 } // end function getBaseSalary
30
31 // calculate earnings
32 double BasePlusCommissionEmployee::earnings() const
33 {
34    // can access protected data of base class
35    return baseSalary + ( commissionRate * grossSales );
36 } // end function earnings
37
38 // print BasePlusCommissionEmployee object
39 void BasePlusCommissionEmployee::print() const
40 {
41    // can access protected data of base class
42    cout << "base-salaried commission employee: " << firstName << ' '
43       << lastName << "\nsocial security number: " << socialSecurityNumber
44       << "\ngross sales: " << grossSales
45       << "\ncommission rate: " << commissionRate
46       << "\nbase salary: " << baseSalary;
47 } // end function print
```

图 12.15　从类 CommissionEmployee 继承 protected 数据的类 BasePlusCommissionEmployee 的实现文件

类BasePlusCommissionEmployee没有继承类CommissionEmployee的构造函数。但是，类BasePlusCommissionEmployee 的构造函数（图 12.15 中第 10~17 行）显式调用类 CommissionEmployee（第14行）的构造函数。回想一下，BasePlusCommissionEmployee的构造函数必须显式调用类CommissionEmployee 的构造函数，因为类 CommissionEmployee 没有包含可以被隐式调用的默认构造函数。

测试修改后的类 BasePlusCommissionEmployee

图 12.9 对第一版的类 BasePlusCommissionEmployee 对象进行测试，图 12.16 则用新版本的类BasePlusCommissionEmployee对象进行相同的测试。注意，这两个程序的输出结果一样。我们在创建第一个类BasePlusCommissionEmployee时没有使用继承，而创建此新版时使用了继承。但是两个类都提供了相同的功能。请注意，这里类BasePlusCommissionEmployee的代码（即头文件和实现文

件）一共74行，明显少于非继承版本的代码（共154行），因为继承版本从类CommissionEmployee中吸收了部分功能，而非继承版本则没有吸收任何功能。而且，现在只有一份声明和定义在类CommissionEmployee 中的这些功能的拷贝，这将使源代码更易于维护、修改和调试，因为与类CommissionEmployee 相关的代码只在图 12.12 和图 12.13 所示的文件中。

```
1  // Fig. 12.16: fig12_16.cpp
2  // Testing class BasePlusCommissionEmployee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 // BasePlusCommissionEmployee class definition
12 #include "BasePlusCommissionEmployee.h"
13
14 int main()
15 {
16    // instantiate BasePlusCommissionEmployee object
17    BasePlusCommissionEmployee
18       employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
19
20    // set floating-point output formatting
21    cout << fixed << setprecision( 2 );
22
23    // get commission employee data
24    cout << "Employee information obtained by get functions: \n"
25       << "\nFirst name is " << employee.getFirstName()
26       << "\nLast name is " << employee.getLastName()
27       << "\nSocial security number is "
28       << employee.getSocialSecurityNumber()
29       << "\nGross sales is " << employee.getGrossSales()
30       << "\nCommission rate is " << employee.getCommissionRate()
31       << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33    employee.setBaseSalary( 1000 ); // set base salary
34
35    cout << "\nUpdated employee information output by print function: \n"
36       << endl;
37    employee.print(); // display the new employee information
38
39    // display the employee's earnings
40    cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41
42    return 0;
43 } // end main
```

```
Employee information obtained by get functions:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information output by print function:

base-salaried commission employee: Bob Lewis
```

图 12.16　派生类可以访问基类的 protected 数据

```
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00
```

图 12.16 派生类可以访问基类的 protected 数据（续）

使用 protected 数据的注意事项

在此例中，我们把基类的数据成员声明为protected，这样派生类就可以直接修改这些数据。因为直接访问 protected 数据成员可以免去调用设置或获取成员函数的开销，所以继承 protected 数据成员会使程序的性能稍稍有所提高。然而，多数情况下，使用private数据成员是更好的软件工程方法，代码优化交给编译器去做就可以了。你的代码将更易于维护、修改和调试。

使用protected数据将产生两个问题。第一，派生类对象不必使用成员函数设置基类的protected数据成员的值。因此，派生类可以很容易地将无效的值赋给基类的protected数据，导致对象处于不可靠的状态中。例如，如果将类CommissionEmployee的数据成员grossSales声明为protected，派生类（如BasePlusCommissionEmployee）的对象可以将一个负值赋给grossSales。第二个使用protected数据成员的问题是：派生类成员函数实现很可能太依赖基类的实现。实际上，派生类应该只依赖基类提供的服务（也就是非private成员函数），而不应该依赖基类的实现。当基类中有protected数据时，如果修改了基类的实现，那么我们同时还需要修改该基类所有的派生类。例如，如果出于某种原因，要将数据成员 firstName 和 lastName 的名字修改为 first 和 last，那么我们将不得不在所有派生类中直接引用这些基类数据成员的地方进行修改。在这种情况下，我们称该程序是“脆弱”的（“fragile”或“brittle”），因为基类中的一个小的改动就可以“破坏”派生类的实现。程序员应该能够修改基类的实现，同时仍能够向派生类提供相同的服务（当然，如果基类提供的服务改变了，我们必须重新实现派生类，好的面向对象设计将会避免这种问题）。



软件工程知识 12.5

如果基类仅向其派生类（和友元）提供服务，而不对其他客户提供该服务，使用protected成员访问说明符是合适的。



软件工程知识 12.6

将基类的数据成员声明为 private（而不是 protected），使程序员可以在不修改派生类实现的同时修改基类的实现。



错误预防技巧 12.1

如果可能，尽量避免在基类中包含有 protected 数据成员。相反，应包含有访问 private 数据成员的成员函数，以保证对象处于可靠状态。

12.4.5 使用 private 数据的 CommissionEmployee ——BasePlusCommission-Employee 继承层次

现在，重新审视一下我们设计的层次，这次我们将使用最好的软件工程实践经验。此刻，类CommissionEmployee（见图 12.17 和图 12.18）声明了 private 数据成员 firstName、lastName、socialSecurityNumber、grossSales 和 commissionRate（见图 12.17 的第 33~37 行），并提供了操作这些数据的 pulbic 成员函数 setFirstName、getFirstName、setLastName、getLastName、setSocialSecurityNumber、getSocialSecurityNumber、setGrossSales、getGrossSales、setCommissionRate、getCommissionRate、earnings 和 print。如果我们决定更改这些数据成员的名称，那么函数 earnings

和print的定义不用修改，只需修改直接操作这些数据成员的获取和设置成员函数即可。请注意，这些修改只发生在基类中，派生类无需丝毫变动。将修改所产生的影响本地化是一条好的软件工程实践经验。派生类BasePlusCommissionEmployee（见图12.19和图12.20）继承了类CommissionEmployee的非 private 成员函数，并且能利用这些函数访问基类的 private 成员。

```
1  // Fig. 12.17: CommissionEmployee.h
2  // CommissionEmployee class definition with good software engineering.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

图 12.17　使用了好的软件工程实践经验的类 CommissionEmployee 的定义

```
1  // Fig. 12.18: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  // constructor
9  CommissionEmployee::CommissionEmployee(
10    const string &first, const string &last, const string &ssn,
11    double sales, double rate )
12    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14    setGrossSales( sales ); // validate and store gross sales
```

图 12.18　类 CommissionEmployee 的实现文件：类 CommissionEmployee 使用成员函数操作其 private 数据

```
15    setCommissionRate( rate ); // validate and store commission rate
16 } // end CommissionEmployee constructor
17
18 // set first name
19 void CommissionEmployee::setFirstName( const string &first )
20 {
21    firstName = first; // should validate
22 } // end function setFirstName
23
24 // return first name
25 string CommissionEmployee::getFirstName() const
26 {
27    return firstName;
28 } // end function getFirstName
29
30 // set last name
31 void CommissionEmployee::setLastName( const string &last )
32 {
33    lastName = last; // should validate
34 } // end function setLastName
35
36 // return last name
37 string CommissionEmployee::getLastName() const
38 {
39    return lastName;
40 } // end function getLastName
41
42 // set social security number
43 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
44 {
45    socialSecurityNumber = ssn; // should validate
46 } // end function setSocialSecurityNumber
47
48 // return social security number
49 string CommissionEmployee::getSocialSecurityNumber() const
50 {
51    return socialSecurityNumber;
52 } // end function getSocialSecurityNumber
53
54 // set gross sales amount
55 void CommissionEmployee::setGrossSales( double sales )
56 {
57    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
58 } // end function setGrossSales
59
60 // return gross sales amount
61 double CommissionEmployee::getGrossSales() const
62 {
63    return grossSales;
64 } // end function getGrossSales
65
66 // set commission rate
67 void CommissionEmployee::setCommissionRate( double rate )
68 {
69    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
70 } // end function setCommissionRate
71
72 // return commission rate
73 double CommissionEmployee::getCommissionRate() const
74 {
75    return commissionRate;
76 } // end function getCommissionRate
77
78 // calculate earnings
```

图 12.18 类 CommissionEmployee 的实现文件：类 CommissionEmployee 使用成员函数操作其 private 数据（续）

```
79 double CommissionEmployee::earnings() const
80 {
81    return getCommissionRate() * getGrossSales();
82 } // end function earnings
83
84 // print CommissionEmployee object
85 void CommissionEmployee::print() const
86 {
87    cout << "commission employee: "
88       << getFirstName() << ' ' << getLastName()
89       << "\nsocial security number: " << getSocialSecurityNumber()
90       << "\ngross sales: " << getGrossSales()
91       << "\ncommission rate: " << getCommissionRate();
92 } // end function print
```

图 12.18　类 CommissionEmployee 的实现文件：类 CommissionEmployee 使用成员函数操作其 private 数据（续）

在 CommissionEmployee 构造函数的实现中（见图 12.18 的第 9~16 行），使用成员初始化器（第 12 行）设置成员 firstName、lastName 和 socialSecurityNumber 的值。我们演示派生类 BasePlusCommissionEmployee（见图 12.19 和图 12.20）调用非 private 基类成员函数(setFirstName、getFirstName、setLastName、getLastName、setSocialSecurityNumber 和 getSocialSecurityNumber)操作这些数据成员的方法。



性能提示 12.2

利用成员函数访问数据成员的值可能比直接访问这些数据稍慢些。但是，如今优化的编译器经过精心设置，可以隐式地执行许多优化工作（如把设置和获取成员函数进行内联）。所以，程序员应该致力于编写出符合软件工程原则的代码，而将优化问题留给编译器去做。一条好的准则是："不要怀疑编译器"。

类 BasePlusCommissionEmployee（见图 12.19 和图 12.20）的成员函数实现（见图 12.20）和其早期版本（见图 12.14 和图 12.15）相比有几处变动。成员函数 earnings（见图 12.20 中第 32~35 行）和 print（第 38~46 行）都调用成员函数 getBaseSalary 得到 baseSalary 的值，而不是直接访问 baseSalary。这样，就将 earnings 和 print 与数据成员 baseSalary 实现的可能变化进行了隔离。例如，如果我们决定重命名数据成员 baseSalary 或者改变其类型，那么只需改动成员函数 setBaseSalary 和 getBaseSalary。

```
1  // Fig. 12.19: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    double earnings() const; // calculate earnings
22    void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

图 12.19　类 BasePlusCommissionEmployee 的头文件

```
1  // Fig. 12.20: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string &first, const string &last, const string &ssn,
12    double sales, double rate, double salary )
13    // explicitly call base-class constructor
14    : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16    setBaseSalary( salary ); // validate and store base salary
17 } // end BasePlusCommissionEmployee constructor
18
19 // set base salary
20 void BasePlusCommissionEmployee::setBaseSalary( double salary )
21 {
22    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // end function setBaseSalary
24
25 // return base salary
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28    return baseSalary;
29 } // end function getBaseSalary
30
31 // calculate earnings
32 double BasePlusCommissionEmployee::earnings() const
33 {
34    return getBaseSalary() + CommissionEmployee::earnings();
35 } // end function earnings
36
37 // print BasePlusCommissionEmployee object
38 void BasePlusCommissionEmployee::print() const
39 {
40    cout << "base-salaried ";
41
42    // invoke CommissionEmployee's print function
43    CommissionEmployee::print();
44
45    cout << "\nbase salary: " << getBaseSalary();
46 } // end function print
```

图 12.20 类 BasePlusCommissionEmployee 继承类 CommissionEmployee，但不能直接访问基类的 private 数据

类 BasePlusCommissionEmployee 的成员函数 earnings（见图 12.20 中的第 32~35 行）重新定义了类 CommissionEmployee 的成员函数 earnings（见图 12.18 中的第 79~82 行），计算带底薪佣金雇员的收入。类 BasePlusCommissionEmployee 的成员函数 earnings 通过表达式 CommissionEmployee::earnings()（见图 12.20 中的第 34 行）调用基类 CommissionEmployee 的 earnings 函数获取雇员收入中提成部分，然后再加上底薪来计算雇员的总收入。请注意由已重新定义基类成员函数的派生类调用基类的函数时的语法格式，即在基类成员函数名前加基类名和二元作用域分辨运算符（::）。这种函数调用是一条好的软件工程实践经验。在“软件工程知识 9.9”中说过，如果一个对象的成员函数能实现另一个对象所需功能，我们应该首选调用这个函数而不是复制其代码。类 BasePlusCommissionEmployee 的 earnings 函数通过调用类 CommissionEmployee 的成员函数 earnings 实现其对象部分收入的计算功能，从而使我们避免了这部分代码的复制，减轻了代码维护的负担。



常见的编程错误 12.2

当派生类重新定义了基类的成员函数时，派生类常常需要调用基类版的该函数做些额外的工作。如果调用基类版的成员函数时忘了在前面加上基类的名称和二元作用域分辨运算符(::)，将导致无限递归的错误，因为派生类版的该成员函数将调用它自己。



常见的编程错误 12.3

如果在派生类中包含一个与基类中同名的但是不同签名的成员函数，那么这个函数会隐藏基类版的那个函数。如果通过派生类的对象的 public 接口试图调用基类版的这个成员函数，将会产生编译错误。

类似地，类 BasePlusCommissionEmployee 的 print 函数（见图 12.20 中的第 38~46 行）重新定义了类 CommissionEmployee 的成员函数 print（见图 12.18 中的第 85~92 行），输出适合于带底薪佣金雇员的信息。类 BasePlusCommissionEmployee 的版本使用受限定的名称 CommissionEmployee::print()（见图 12.20 中的第 43 行），调用类 CommissionEmployee 的 print 成员函数，显示类 BasePlusCommissionEmployee 对象的部分信息（即，字符串 "commission employee" 和类 CommissionEmployee 的 private 数据成员的值）。类 BasePlusCommissionEmployee 的 print 函数输出类 CommissionEmployee 对象中的其余信息（即，类 BasePlusCommissionEmployee 的基本工资值）。

图 12.21 对类 BasePlusCommissionEmployee 的对象执行的操作，与图 12.9 和图 12.16 对类 CommissionEmployee 和类 BasePlusCommissionEmployee 的对象分别所做的操作是一样的。尽管每个“带底薪佣金雇员”类的行为都相同，但是，这一节的类 BasePlusCommissionEmployee 遵循了最好的软件工程规范。我们使用继承和成员函数，隐藏数据，保持一致性，建立了一个遵循良好软件工程规范的类。

```
1  // Fig. 12.21: fig12_21.cpp
2  // Testing class BasePlusCommissionEmployee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 // BasePlusCommissionEmployee class definition
12 #include "BasePlusCommissionEmployee.h"
13
14 int main()
15 {
16    // instantiate BasePlusCommissionEmployee object
17    BasePlusCommissionEmployee
18       employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
19
20    // set floating-point output formatting
21    cout << fixed << setprecision( 2 );
22
23    // get commission employee data
24    cout << "Employee information obtained by get functions: \n"
25       << "\nFirst name is " << employee.getFirstName()
26       << "\nLast name is " << employee.getLastName()
27       << "\nSocial security number is "
28       << employee.getSocialSecurityNumber()
29       << "\nGross sales is " << employee.getGrossSales()
30       << "\nCommission rate is " << employee.getCommissionRate()
31       << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33    employee.setBaseSalary( 1000 ); // set base salary
```

图 12.21　派生类通过继承而来的 public 或 protected 成员函数可以访问基类的 private 数据

```
34
35     cout << "\nUpdated employee information output by print function: \n"
36        << endl;
37     employee.print(); // display the new employee information
38
39     // display the employee's earnings
40     cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41
42     return 0;
43  } // end main
```

```
Employee information obtained by get functions:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information output by print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00
```

图 12.21 派生类通过继承而来的 public 或 protected 成员函数可以访问基类的 private 数据（续）

在本节中，有一套精心设计的例子，讲解了使用继承的良好软件工程的要点。大家学习了如何通过继承创建派生类，如何使用protected基类成员使派生类可以访问继承而来的基类数据成员，如何重新定义基类的成员函数而为派生类的对象提供更适合的版本。此外，还学习了如何应用第9章中的软件工程技术，创建易于维护、修改和调试的类。

12.5 派生类中的构造函数和析构函数

正如前面所述，实例化派生类对象启动了一连串构造函数的调用，其中派生类的构造函数在执行它自己的任务之前，先显式调用（通过基类成员的初始化器）或隐式（调用基类的默认构造函数）地调用其直接基类的构造函数。同样，如果该基类也是从别的类派生而来的，则该基类构造函数需调用类层次中直接在该基类上的类的构造函数，等等。在这个构造函数调用链中调用的最后一个构造函数是在继承层次中最顶层的构造函数，但是实际上它的函数体是最先执行完毕的，最早调用的派生类构造函数最晚完成其函数体的执行。每个基类构造函数初始化派生类对象继承的该基类的数据成员。例如，考虑图 12.17 至图 12.20 中的 CommissionEmployee/BasePlusCommissionEmployee 的层次。当程序创建类 BasePlusCommissionEmployee 的一个对象时，将调用类 CommissionEmployee 的构造函数。因为类 CommissionEmployee 是继承层次中的基类，所以它的构造函数执行，所初始化的CommissionEmployee的private数据成员同时也是类BasePlusCommissionEmployee对象的一部分。当类CommissionEmployee的构造函数执行完毕时，它将程序控制权还给类BasePlusCommissionEmployee 的构造函数，后者然后初始化这个 BasePlusCommissionEmployee 类对象的 baseSalary。



软件工程知识 12.7

当程序创建一个派生类对象时，派生类构造函数立即调用基类的构造函数，基类的构造函数执行，然后派生类的成员初始化器执行，最后派生类构造函数的函数体执行。如果这时的继承层次有多层，那么这一过程还将向上翻滚。

当销毁派生类的对象时，程序将调用对象的析构函数。这又将展开一连串的析构函数调用，其中派生类、派生类的直接基类、派生类的间接基类以及类的成员的析构函数会按照它们构造函数执行次序的相反顺序依次执行。当调用派生类对象的析构函数时，该析构函数执行其任务，然后调用继承层次中上一层基类的析构函数，这一过程重复进行，直到顶层的最后一个基类的析构函数被调用。之后，该对象就从内存中删除了。



软件工程知识 12.8

假设我们创建一个派生类对象，这个派生类及其基类中都包含其他类的对象。当这个派生类的对象被创建时，首先执行基类成员对象的构造函数，然后执行基类的构造函数，接着执行派生类成员对象的构造函数，最后执行派生类的构造函数。派生类对象析构函数的调用顺序与相应的构造函数的调用顺序正好相反。

派生类不会继承基类的构造函数、析构函数和重载的赋值运算符（参见第 11 章）。但是，派生类的构造函数、析构函数和重载的赋值运算符可以调用基类的构造函数、析构函数和重载的赋值运算符函数。

接下来的例子中，我们将重新讨论佣金雇员的继承层次，定义类 CommissionEmployee（见图 12.22 和图 12.23）和类 BasePlusCommissionEmployee（见图 12.24 和图 12.25），使它们包含相应的构造函数和析构函数，并且这些函数在调用时都会打印相关信息。如图 12.26 中的输出所示，这些信息展示了在继承层次中对象的构造函数和析构函数的调用顺序。

```
1  // Fig. 12.22: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14    ~CommissionEmployee(); // destructor
15
16    void setFirstName( const string & ); // set first name
17    string getFirstName() const; // return first name
18
19    void setLastName( const string & ); // set last name
20    string getLastName() const; // return last name
21
22    void setSocialSecurityNumber( const string & ); // set SSN
23    string getSocialSecurityNumber() const; // return SSN
24
25    void setGrossSales( double ); // set gross sales amount
26    double getGrossSales() const; // return gross sales amount
27
28    void setCommissionRate( double ); // set commission rate
29    double getCommissionRate() const; // return commission rate
30
31    double earnings() const; // calculate earnings
32    void print() const; // print CommissionEmployee object
33 private:
34    string firstName;
35    string lastName;
36    string socialSecurityNumber;
37    double grossSales; // gross weekly sales
38    double commissionRate; // commission percentage
39 }; // end class CommissionEmployee
40
41 #endif
```

图 12.22　类 CommissionEmployee 的头文件

```
1  // Fig. 12.23: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "CommissionEmployee.h" // CommissionEmployee class definition
8
9  // constructor
10 CommissionEmployee::CommissionEmployee(
11    const string &first, const string &last, const string &ssn,
12    double sales, double rate )
13    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
14 {
15    setGrossSales( sales ); // validate and store gross sales
16    setCommissionRate( rate ); // validate and store commission rate
17
18    cout << "CommissionEmployee constructor: " << endl;
19    print();
20    cout << "\n\n";
21 } // end CommissionEmployee constructor
22
23 // destructor
24 CommissionEmployee::~CommissionEmployee()
25 {
26    cout << "CommissionEmployee destructor: " << endl;
27    print();
28    cout << "\n\n";
29 } // end CommissionEmployee destructor
30
31 // set first name
32 void CommissionEmployee::setFirstName( const string &first )
33 {
34    firstName = first; // should validate
35 } // end function setFirstName
36
37 // return first name
38 string CommissionEmployee::getFirstName() const
39 {
40    return firstName;
41 } // end function getFirstName
42
43 // set last name
44 void CommissionEmployee::setLastName( const string &last )
45 {
46    lastName = last; // should validate
47 } // end function setLastName
48
49 // return last name
50 string CommissionEmployee::getLastName() const
51 {
52    return lastName;
53 } // end function getLastName
54
55 // set social security number
56 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
57 {
58    socialSecurityNumber = ssn; // should validate
59 } // end function setSocialSecurityNumber
60
61 // return social security number
62 string CommissionEmployee::getSocialSecurityNumber() const
63 {
64    return socialSecurityNumber;
```

图 12.23　类 CommissionEmployee 的构造函数输出信息

```
65  } // end function getSocialSecurityNumber
66
67  // set gross sales amount
68  void CommissionEmployee::setGrossSales( double sales )
69  {
70     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
71  } // end function setGrossSales
72
73  // return gross sales amount
74  double CommissionEmployee::getGrossSales() const
75  {
76     return grossSales;
77  } // end function getGrossSales
78
79  // set commission rate
80  void CommissionEmployee::setCommissionRate( double rate )
81  {
82     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
83  } // end function setCommissionRate
84
85  // return commission rate
86  double CommissionEmployee::getCommissionRate() const
87  {
88     return commissionRate;
89  } // end function getCommissionRate
90
91  // calculate earnings
92  double CommissionEmployee::earnings() const
93  {
94     return getCommissionRate() * getGrossSales();
95  } // end function earnings
96
97  // print CommissionEmployee object
98  void CommissionEmployee::print() const
99  {
100    cout << "commission employee: "
101       << getFirstName() << ' ' << getLastName()
102       << "\nsocial security number: " << getSocialSecurityNumber()
103       << "\ngross sales: " << getGrossSales()
104       << "\ncommission rate: " << getCommissionRate();
105 } // end function print
```

图 12.23　类 CommissionEmployee 的构造函数输出信息（续）

```
1  // Fig. 12.24: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17    ~BasePlusCommissionEmployee(); // destructor
18
19    void setBaseSalary( double ); // set base salary
20    double getBaseSalary() const; // return base salary
21
22    double earnings() const; // calculate earnings
23    void print() const; // print BasePlusCommissionEmployee object
```

图 12.24　类 BasePlusCommissionEmployee 的头文件

```
24 private:
25    double baseSalary; // base salary
26 }; // end class BasePlusCommissionEmployee
27
28 #endif
```

图 12.24 类 BasePlusCommissionEmployee 的头文件（续）

```
1  // Fig. 12.25: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // BasePlusCommissionEmployee class definition
8  #include "BasePlusCommissionEmployee.h"
9
10 // constructor
11 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
12    const string &first, const string &last, const string &ssn,
13    double sales, double rate, double salary )
14    // explicitly call base-class constructor
15    : CommissionEmployee( first, last, ssn, sales, rate )
16 {
17    setBaseSalary( salary ); // validate and store base salary
18
19    cout << "BasePlusCommissionEmployee constructor: " << endl;
20    print();
21    cout << "\n\n";
22 } // end BasePlusCommissionEmployee constructor
23
24 // destructor
25 BasePlusCommissionEmployee::~BasePlusCommissionEmployee()
26 {
27    cout << "BasePlusCommissionEmployee destructor: " << endl;
28    print();
29    cout << "\n\n";
30 } // end BasePlusCommissionEmployee destructor
31
32 // set base salary
33 void BasePlusCommissionEmployee::setBaseSalary( double salary )
34 {
35    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
36 } // end function setBaseSalary
37
38 // return base salary
39 double BasePlusCommissionEmployee::getBaseSalary() const
40 {
41    return baseSalary;
42 } // end function getBaseSalary
43
44 // calculate earnings
45 double BasePlusCommissionEmployee::earnings() const
46 {
47    return getBaseSalary() + CommissionEmployee::earnings();
48 } // end function earnings
49
50 // print BasePlusCommissionEmployee object
51 void BasePlusCommissionEmployee::print() const
52 {
53    cout << "base-salaried ";
54
55    // invoke CommissionEmployee's print function
56    CommissionEmployee::print();
57
58    cout << "\nbase salary: " << getBaseSalary();
59 } // end function print
```

图 12.25 类 BasePlusCommissionEmployee 的构造函数输出信息

在本例中，我们修改了类 CommissionEmployee 的构造函数（图 12.23 中的第 10~21 行），并添加了类 CommissionEmployee 的析构函数（第 24~29 行），这两个函数调用时都会输出一段关于其调用的文字信息。我们还修改了类 BasePlusCommissionEmployee 的构造函数（见图 12.25 中的第 11~22 行），并添加了类 BasePlusCommissionEmployee 的析构函数（第 25~30 行），同样，它们都将输出关于其调用的信息。

图 12.26 演示了作为继承层次一部分的类对象其构造函数和析构函数的调用顺序。main 函数（第 15~34 行）首先在一个独立的语句块（见第 20~23 行）中实例化了 CommissionEmployee 对象 employee1（第 21~22 行）。这个对象迅速地进出作用域（一旦创建完对象就到了语句块的末尾），所以 CommissionEmployee 的构造函数和析构函数都被调用了。接下来，第 26~27 行实例化类 BasePlusCommissionEmployee 的对象 employee2。这将调用类 CommissionEmployee 的构造函数，用 BasePlusCommissionEmployee 传递的值进行输出显示，然后再执行 BasePlusCommissionEmployee 构造函数中指定的输出。第 30~31 行实例化了类 BasePlusCommissionEmployee 的对象 employee3。同样，类 CommissionEmployee 和 BasePlusCommissionEmployee 的构造函数又被依次调用。请注意，在 employee2 和 employee3 的创建中，CommissionEmployee 构造函数的函数体的执行总是先于 BasePlusCommissionEmployee 构造函数的函数体的执行。当 main 函数的执行遇到结束花括号时，将调用对象 employee2 和 employee3 的析构函数。但是，析构函数的调用顺序和构造函数的调用顺序刚好相反，所以先为对象 employee3 依次调用 BasePlusCommissionEmployee 析构函数和 CommissionEmployee 析构函数。然后再为类对象 employee2 依次调用 BasePlusCommissionEmployee 析构函数和 CommissionEmployee 析构函数。

```
1  // Fig. 12.26: fig12_26.cpp
2  // Display order in which base-class and derived-class constructors
3  // and destructors are called.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 // BasePlusCommissionEmployee class definition
13 #include "BasePlusCommissionEmployee.h"
14
15 int main()
16 {
17    // set floating-point output formatting
18    cout << fixed << setprecision( 2 );
19
20    { // begin new scope
21       CommissionEmployee employee1(
22          "Bob", "Lewis", "333-33-3333", 5000, .04 );
23    } // end scope
24
25    cout << endl;
26    BasePlusCommissionEmployee
27       employee2( "Lisa", "Jones", "555-55-5555", 2000, .06, 800 );
28
29    cout << endl;
30    BasePlusCommissionEmployee
31       employee3( "Mark", "Sands", "888-88-8888", 8000, .15, 2000 );
32    cout << endl;
33    return 0;
34 } // end main
```

图 12.26　构造函数和析构函数的调用顺序

```
CommissionEmployee constructor:
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04

CommissionEmployee destructor:
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04

CommissionEmployee constructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06

BasePlusCommissionEmployee constructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 800.00

CommissionEmployee constructor:
commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15

BasePlusCommissionEmployee constructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 2000.00

BasePlusCommissionEmployee destructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 2000.00

CommissionEmployee destructor:
commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15

BasePlusCommissionEmployee destructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 800.00

CommissionEmployee destructor:
commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
```

图 12.26　构造函数和析构函数的调用顺序（续）

12.6　public、protected 和 private 继承

当由基类派生出一个类时，继承基类的方式有三种，即 public 继承、protected 继承和 private 继承。实际上，一般很难采用 private 继承和 protected 继承，而且使用时需十分小心。在本书中，我

们只采用 public 继承。图 12.27 总结了不同方式的继承中派生类对基类成员的访问权限，其中第一列是基类成员访问说明符。

基类成员访问说明符	继承类型		
	public继承	protected继承	private继承
public	在派生类中为public 可以由成员函数、friend函数和非成员函数直接访问	在派生类中为protected 可以由成员函数和friend函数直接访问	在派生类中为private 可以由成员函数和friend函数直接访问
protected	在派生类中为protected 可以由成员函数和friend函数直接访问	在派生类中为protected 可以由成员函数和friend函数直接访问	在派生类中为private 可以由成员函数和friend函数直接访问
private	在派生类中隐藏 可以通过基类的public或protected成员函数由成员函数和friend函数间接访问	在派生类中隐藏 可以通过基类的public或protected成员函数由成员函数和friend函数间接访问	在派生类中隐藏 可以通过基类的public或protected成员函数由成员函数和friend函数间接访问

图 12.27　派生类对基类成员的访问权限

当从public基类派生一个类时，基类的public成员成为派生类中的public成员，基类的protected成员成为派生类中的 protected 成员。派生类永远不能直接访问基类的 private 成员，但是可以通过调用基类的 public 和 protected 成员进行间接访问。

当从protected基类派生一个类时，基类的public和protected成员都变成派生类中的protected成员。当从基类派生一个类时，基类的 public 和 protected 成员都变成派生类的 private 成员（例如，基类的这些函数都成为派生类的函数）。private 和 protected 继承不是“is-a”关系。

12.7　关于继承的软件工程

在本节中，我们将讨论使用继承定制现有软件。如图 12.27 所示，当我们使用继承从现有类创建新类时，新类继承了现有类的数据成员和成员函数。我们可以通过添加另外的成员或者重新定义基类的成员定制这个新类，使它满足需求。C++程序员不用访问基类的源代码即可完成对派生类的这项任务，但是派生类必须能够链接到基类的目标代码。这种强大的功能非常吸引独立软件供应商（ISV）。ISV 可以开发具有销售所有权或发布许可证的类，然后以目标代码的形式将这些类提供给用户使用。这样，用户不用直接访问ISV专有的源代码就能快速地从类库中继承并创建新类。所有的 ISV 需要随目标代码提供相应的头文件。

有时候，学生很难认识到大型软件项目的设计者所面临的问题。有项目开发经验的人都知道有效的软件重用能够加快软件开发的进程。面向对象编程鼓励软件重用，以缩短软件开发时间并提高软件的质量。

正是通过继承机制，才使实用的类库发挥出了软件重用的最大优势。我们看到，随着个人电脑的出现，软件行业迅速发展起来，独立软件供应商生产的套装软件日益增长。现在，继承更促使他们纷纷投入到类库的创建和销售热潮中。应用程序设计者用类库构建他们的应用程序，同时类库设

13.3 类继承层次中对象之间的关系

在12.4节创建的雇员类层次结构中，类BasePlusCommissionEmployee继承了类CommissionEmployee。第12章的例子通过类 CommissionEmployee 和 BasePlusCommissionEmployee 的对象名调用其成员函数来操纵这些对象。现在，我们将更详细地分析在继承层次结构中各个类之间的关系。下面几节提供了一系列实例，演示如何将基类指针和派生类指针指向基类对象和派生类对象，以及如何利用这些指针调用操纵这些对象的成员函数。这一节的最后，我们将演示如何通过指向派生类对象的基类指针来获得多态性行为。

在 13.3.1 节，我们把派生类对象的地址赋给基类指针。然后，展示通过该基类指针调用函数，从而调用基类的功能——也就是句柄类型决定哪个函数被调用。在 13.3.2 节，我们将基类对象的地址赋给派生类指针，结果将导致编译错误。我们讨论这一错误消息，并研究编译器不允许这样赋值的原因。在 13.3.3 节，我们把派生类对象的地址赋给基类指针，然后分析为什么利用该基类指针只能调用基类的功能，而当通过该基类指针试图调用派生类的成员函数时，编译产生了错误。最后，在 13.3.4 节，我们介绍 virtual 函数和多态性，这是通过将基类函数声明为 virtual 而实现的。然后，我们把派生类对象的地址赋给基类指针，并利用这一指针调用派生类的功能——这恰恰是我们需要达到多态性行为的功能。

演示这些例子就是想说明这样一个重要思想：派生类的对象可以被当做它的基类对象的处理。这导致了一些有趣的操作。例如，在程序中可以创建一个基类指针的数组，数组元素指向许多派生类的对象。尽管事实上派生类对象和基类对象是不同的类型，但是编译器允许这种赋值，因为每个派生类对象都是一个基类对象。然而，我们却不能把基类对象当做任何一个派生类的对象来处理。例如，在第12章定义的类继承层次中，类CommissionEmployee的对象就不是类BasePlusCommissionEmployee 的对象，因为类 CommissionEmployee 的对象没有数据成员 baseSalary 和成员函数 setBaseSalary 和 getBaseSalary。“是一个”的关系只适用于从派生类到它的直接或者间接基类。

13.3.1 从派生类对象调用基类函数

图 13.1 到图 13.5 中的实例演示了将基类指针和派生类指针指向基类对象和派生类对象的三种方式。前两种是简单直接的——将基类指针指向基类对象（并调用基类的功能），以及将派生类指针指向派生类对象（并调用派生类的功能）。然后，我们通过把基类指针指向派生类对象，以及展示基类的功能性的确在派生类对象中是可用的，来说明派生类和基类之间的关系（即在继承层次中的“是一个”的关系）。

我们在第12章中讨论过的CommissionEmployee类（见图 13.1 和图 13.2），用于表示根据销售业绩提成的一类雇员。我们在第12章还讨论过BasePlusCommissionEmployee类（见图 13.3 和图 13.4），用于表示有基本工资但同时也根据销售业绩提成的一类雇员。每个类BasePlusCommissionEmployee 的对象都是一个类 CommissionEmployee 的对象，但他还有基本工资。

```
1  // Fig. 13.1: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
```

图 13.1 CommissionEmployee 类的头文件

```
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

图 13.1　CommissionEmployee 类的头文件（续）

```
1  // Fig. 13.2: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  // constructor
9  CommissionEmployee::CommissionEmployee(
10    const string &first, const string &last, const string &ssn,
11    double sales, double rate )
12    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14    setGrossSales( sales ); // validate and store gross sales
15    setCommissionRate( rate ); // validate and store commission rate
16 } // end CommissionEmployee constructor
17
18 // set first name
19 void CommissionEmployee::setFirstName( const string &first )
20 {
21    firstName = first; // should validate
22 } // end function setFirstName
23
24 // return first name
25 string CommissionEmployee::getFirstName() const
26 {
27    return firstName;
28 } // end function getFirstName
29
```

图 13.2　CommissionEmployee 类的实现文件

```
30 // set last name
31 void CommissionEmployee::setLastName( const string &last )
32 {
33    lastName = last;  // should validate
34 } // end function setLastName
35
36 // return last name
37 string CommissionEmployee::getLastName() const
38 {
39    return lastName;
40 } // end function getLastName
41
42 // set social security number
43 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
44 {
45    socialSecurityNumber = ssn; // should validate
46 } // end function setSocialSecurityNumber
47
48 // return social security number
49 string CommissionEmployee::getSocialSecurityNumber() const
50 {
51    return socialSecurityNumber;
52 } // end function getSocialSecurityNumber
53
54 // set gross sales amount
55 void CommissionEmployee::setGrossSales( double sales )
56 {
57    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
58 } // end function setGrossSales
59
60 // return gross sales amount
61 double CommissionEmployee::getGrossSales() const
62 {
63    return grossSales;
64 } // end function getGrossSales
65
66 // set commission rate
67 void CommissionEmployee::setCommissionRate( double rate )
68 {
69    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
70 } // end function setCommissionRate
71
72 // return commission rate
73 double CommissionEmployee::getCommissionRate() const
74 {
75    return commissionRate;
76 } // end function getCommissionRate
77
78 // calculate earnings
79 double CommissionEmployee::earnings() const
80 {
81    return getCommissionRate() * getGrossSales();
82 } // end function earnings
83
84 // print CommissionEmployee object
85 void CommissionEmployee::print() const
86 {
87    cout << "commission employee: "
88       << getFirstName() << ' ' << getLastName()
89       << "\nsocial security number: " << getSocialSecurityNumber()
90       << "\ngross sales: " << getGrossSales()
91       << "\ncommission rate: " << getCommissionRate();
92 } // end function print
```

图 13.2　CommissionEmployee 类的实现文件（续）

```
// Fig. 13.3: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class
// CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
using std::string;

#include "CommissionEmployee.h" // CommissionEmployee class declaration

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
   BasePlusCommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0, double = 0.0 );

   void setBaseSalary( double ); // set base salary
   double getBaseSalary() const; // return base salary

   double earnings() const; // calculate earnings
   void print() const; // print BasePlusCommissionEmployee object
private:
   double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

#endif
```

图 13.3　BasePlusCommissionEmployee 类的头文件

```
// Fig. 13.4: BasePlusCommissionEmployee.cpp
// Class BasePlusCommissionEmployee member-function definitions.
#include <iostream>
using std::cout;

// BasePlusCommissionEmployee class definition
#include "BasePlusCommissionEmployee.h"

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string &first, const string &last, const string &ssn,
   double sales, double rate, double salary )
   // explicitly call base-class constructor
   : CommissionEmployee( first, last, ssn, sales, rate )
{
   setBaseSalary( salary ); // validate and store base salary
} // end BasePlusCommissionEmployee constructor

// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
   baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
} // end function setBaseSalary

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
   return baseSalary;
} // end function getBaseSalary

// calculate earnings
double BasePlusCommissionEmployee::earnings() const
```

图 13.4　BasePlusCommissionEmployee 类的实现文件

```
33 {
34    return getBaseSalary() + CommissionEmployee::earnings();
35 } // end function earnings
36
37 // print BasePlusCommissionEmployee object
38 void BasePlusCommissionEmployee::print() const
39 {
40    cout << "base-salaried ";
41
42    // invoke CommissionEmployee's print function
43    CommissionEmployee::print();
44
45    cout << "\nbase salary: " << getBaseSalary();
46 } // end function print
```

图 13.4 BasePlusCommissionEmployee 类的实现文件（续）

BasePlusCommissionEmployee 类的 earnings 成员函数（见图 13.4 中的第 32~35 行）通过加入基本工资重新定义了 CommissionEmployee 类的 earnings 成员函数（见图 13.2 中的第 79~82 行）。BasePlusCommissionEmployee 类的 print 成员函数（见图 13.4 中的第 38~46 行）重新定义了 CommissionEmployee 类的 print 成员函数（见图 13.2 中的第 85~92 行），它不仅与 CommissionEmployee 类的 print 函数输出一样的信息，还显示了雇员的基本工资。

图 13.5 中，在第 19~20 行创建了一个 CommissionEmployee 类对象，在第 23 行创建了一个指向 CommissionEmployee 类对象的指针，在第 26~27 行创建了一个 BasePlusCommissionEmployee 类对象，并在第 30 行创建了一个指向 BasePlusCommissionEmployee 类对象的指针。在第 37~39 行用每个对象的名字（分别是 commissionEmployee 和 basePlusCommissionEmployee）调用它的 print 成员函数。第 42 行把基类对象 commissionEmployee 的地址赋给了基类指针 commissionEmployeePtr，第 45 行则利用该指针调用这个 CommissionEmployee 类对象的成员函数 print。此时所调用的 print 函数是在基类 CommissionEmployee 中定义的版本。同样，第 48 行把派生类对象 basePlusCommissionEmployee 的地址赋给派生类指针 basePlusCommissionEmployeePtr，而第 52 行利用该指针调用这个 BasePlusCommissionEmployee 类对象的成员函数 print，此时所调用的 print 函数是在派生类 BasePlusCommissionEmployee 中定义的版本。然后，第 55 行把派生类对象 basePlusCommissionEmployee 的地址赋给了基类指针 commissionEmployeePtr，在第 59 行，利用此指针调用成员函数 print。C++ 编译器允许这一“交叉赋值”，是因为每个派生类对象都是一个基类对象。请注意，虽然事实上基类 CommissionEmployee 的指针指向了派生类 BasePlusCommissionEmployee 的对象，但调用的却仍是基类 CommissionEmployee 的成员函数 print（而不是类 BasePlusCommissionEmployee 的成员函数 print）。

```
 1 // Fig. 13.5: fig13_05.cpp
 2 // Aiming base-class and derived-class pointers at base-class
 3 // and derived-class objects, respectively.
 4 #include <iostream>
 5 using std::cout;
 6 using std::endl;
 7 using std::fixed;
 8
 9 #include <iomanip>
10 using std::setprecision;
11
12 // include class definitions
13 #include "CommissionEmployee.h"
14 #include "BasePlusCommissionEmployee.h"
15
16 int main()
```

图 13.5 将基类和派生类对象的地址赋给基类和派生类的指针

```
{
   // create base-class object
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );

   // create base-class pointer
   CommissionEmployee *commissionEmployeePtr = 0;

   // create derived-class object
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

   // create derived-class pointer
   BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

   // output objects commissionEmployee and basePlusCommissionEmployee
   cout << "Print base-class and derived-class objects:\n\n";
   commissionEmployee.print(); // invokes base-class print
   cout << "\n\n";
   basePlusCommissionEmployee.print(); // invokes derived-class print

   // aim base-class pointer at base-class object and print
   commissionEmployeePtr = &commissionEmployee; // perfectly natural
   cout << "\n\n\nCalling print with base-class pointer to "
      << "\nbase-class object invokes base-class print function:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print

   // aim derived-class pointer at derived-class object and print
   basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
   cout << "\n\n\nCalling print with derived-class pointer to "
      << "\nderived-class object invokes derived-class "
      << "print function:\n\n";
   basePlusCommissionEmployeePtr->print(); // invokes derived-class print

   // aim base-class pointer at derived-class object and print
   commissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\n\nCalling print with base-class pointer to "
      << "derived-class object\ninvokes base-class print "
      << "function on that derived-class object:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print
   cout << endl;
   return 0;
} // end main
```

```
Print base-class and derived-class objects:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Calling print with base-class pointer to
base-class object invokes base-class print function:
```

图 13.5　将基类和派生类对象的地址赋给基类和派生类的指针（续）

```
commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06


Calling print with derived-class pointer to
derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Calling print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object:

commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
```

图 13.5 将基类和派生类对象的地址赋给基类和派生类的指针（续）

程序中每个成员函数 print 调用的输出揭示了所调用的功能是由用于调用函数的句柄（即指针或引用）的类型决定的，而不是由句柄所指向的对象的类型决定。在 13.3.4 节我们介绍 virtual 函数时，将演示不调用句柄类型的功能，而调用对象类型的功能是可能的。我们将会看到这对于实现多态性行为是至关重要的——这是本章关键的主题。

13.3.2 将派生类指针指向基类对象

在13.3.1节，我们把派生类对象的地址赋给了基类指针，并解释了C++编译器允许这种赋值的原因：每个派生类对象都是一个基类对象。在图13.6中我们采用相反的做法，将派生类指针指向基类对象。注意：图13.6中的程序使用了图13.1到图13.4中定义的CommissionEmployee类和BasePlusCommissionEmployee类。图13.6中，第8~9行创建了一个CommissionEmployee类对象，并在第10行创建了一个BasePlusCommissionEmployee类的指针。第14行试图将基类对象commissionEmployee的地址赋给派生类指针basePlusCommissionEmployeePtr，但是，C++编译器产生错误信息。编译器不允许这一赋值，因为CommissionEmployee类对象并不是BasePlusCommissionEmployee类对象。如果编译器允许这种赋值，考虑一下会产生什么后果。通过一个BasePlusCommissionEmployee类指针，我们可以为它所指向的对象（即基类对象commissionEmployee）调用类BasePlusCommissionEmployee的每个成员函数，包括成员函数setBaseSalary。然而，CommissionEmployee对象既没有提供一个成员函数setBaseSalary，也没有提供要设置的一个数据成员baseSalary。这将导致一些问题，因为成员函数setBaseSalary会假定有一个baseSalary数据成员要设置在它在BasePlusCommissionEmployee类对象中的“惯常位置”上。但是，这一内存地址并不属于这个CommissionEmployee对象，因此成员函数setBaseSalary可能覆盖内存中其他重要的数据，这很可能是不同对象中的数据。

13.3.3 通过基类指针调用派生类的成员函数

利用基类指针，编译器只允许我们调用基类的成员函数。因此，如果一个基类指针指向了一个派生类对象，并且试图访问只在派生类中拥有的成员函数，那么就会产生编译错误。

```
1  // Fig. 13.6: fig13_06.cpp
2  // Aiming a derived-class pointer at a base-class object.
3  #include "CommissionEmployee.h"
4  #include "BasePlusCommissionEmployee.h"
5
6  int main()
7  {
8     CommissionEmployee commissionEmployee(
9        "Sue", "Jones", "222-22-2222", 10000, .06 );
10    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
11
12    // aim derived-class pointer at base-class object
13    // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
14    basePlusCommissionEmployeePtr = &commissionEmployee;
15    return 0;
16 } // end main
```

Borland C++ command-line compiler error messages:

```
Error E2034 Fig13_06\fig13_06.cpp 14: Cannot convert 'CommissionEmployee *'
   to 'BasePlusCommissionEmployee *' in function main()
```

GNU C++ compiler error messages:

```
fig13_06.cpp:14: error: invalid conversion from `CommissionEmployee*' to
   `BasePlusCommissionEmployee*'
```

Microsoft Visual C++.NET compiler error messages:

```
C:\scpphtp5_examples\ch13\Fig13_06\fig13_06.cpp(14) : error C2440:
   '=' : cannot convert from 'CommissionEmployee *__w64 ' to
   'BasePlusCommissionEmployee *'
         Cast from base to derived requires dynamic_cast or static_cast
```

图 13.6　将派生类指针指向基类对象

图 13.7 演示了试图利用基类指针调用派生类成员函数的后果。注意：我们仍然使用图 13.1 到图 13.4 中定义的 CommissionEmployee 类和 BasePlusCommissionEmployee 类。第 9 行创建了一个指向 CommissionEmployee 类对象的指针 commissionEmployeePtr，第 10~11 行创建了一个 BasePlusCommissionEmployee 类对象。第 14 行使指针 commissionEmployeePtr 指向派生类对象 basePlusCommissionEmployee。回忆 13.3.1 节，C++ 编译器允许这种赋值，因为每个 BasePlusCommissionEmployee 类对象都是一个 CommissionEmployee 类对象（换言之，每个 BasePlusCommissionEmployee 类对象包含了 CommissionEmployee 类对象的所有功能）。第 18~22 行利用这个基类指针调用了基类的成员函数 getFirstName、getLastName、getSocialSecurityNumber、getGrossSales 和 getCommissionRate。所有这些调用都是合法的，因为 BasePlusCommissionEmployee 类从 CommissionEmployee 类继承了所有这些成员函数。我们知道 commissionEmployeePtr 已指向了一个 BasePlusCommissionEmployee 类对象，所以在第 26~27 行，我们试图调用 BasePlusCommissionEmployee 类的成员函数 getBaseSalary 和 setBaseSalary。C++ 编译器在这两行上都产生了错误信息，因为这两个函数都不是基类 CommissionEmployee 的成员函数。该句柄只能调用与它关联的类类型的成员函数（在本例中，通过一个 CommissionEmployee 类指针，我们只能调用 CommissionEmployee 类的成员函数 setFirstName、getFirstName、setLastName、getLastName、setSocialSecurityNumber、

getSocialSecurityNumber、setGrossSales、getGrossSales、setCommissionRate、getCommissionRate、earnings 和 print。

```
1  // Fig. 13.7: fig13_07.cpp
2  // Attempting to invoke derived-class-only member functions
3  // through a base-class pointer.
4  #include "CommissionEmployee.h"
5  #include "BasePlusCommissionEmployee.h"
6
7  int main()
8  {
9     CommissionEmployee *commissionEmployeePtr = 0; // base class
10    BasePlusCommissionEmployee basePlusCommissionEmployee(
11       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class
12
13    // aim base-class pointer at derived-class object
14    commissionEmployeePtr = &basePlusCommissionEmployee;
15
16    // invoke base-class member functions on derived-class
17    // object through base-class pointer
18    string firstName = commissionEmployeePtr->getFirstName();
19    string lastName = commissionEmployeePtr->getLastName();
20    string ssn = commissionEmployeePtr->getSocialSecurityNumber();
21    double grossSales = commissionEmployeePtr->getGrossSales();
22    double commissionRate = commissionEmployeePtr->getCommissionRate();
23
24    // attempt to invoke derived-class-only member functions
25    // on derived-class object through base-class pointer
26    double baseSalary = commissionEmployeePtr->getBaseSalary();
27    commissionEmployeePtr->setBaseSalary( 500 );
28    return 0;
29 } // end main
```

Borland C++ command-line compiler error messages:

```
Error E2316 Fig13_07\fig13_07.cpp 26: 'getBaseSalary' is not a member of
   'CommissionEmployee' in function main()
Error E2316 Fig13_07\fig13_07.cpp 27: 'setBaseSalary' is not a member of
   'CommissionEmployee' in function main()
```

Microsoft Visual C++.NET compiler error messages:

```
C:\scpphtp5_examples\ch13\Fig13_07\fig13_07.cpp(26) : error C2039:
   'getBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp5_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
C:\scpphtp5_examples\ch13\Fig13_07\fig13_07.cpp(27) : error C2039:
   'setBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp5_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
```

GNU C++ compiler error messages:

```
fig13_07.cpp:26: error: `getBaseSalary' undeclared (first use this function)
fig13_07.cpp:26: error: (Each undeclared identifier is reported only once for
   each function it appears in.)
fig13_07.cpp:27: error: `setBaseSalary' undeclared (first use this function)
```

图 13.7　试图通过基类指针调用只在派生类中拥有的函数



错误预防技巧 13.1

当程序员浏览类的层次结构以定位要重用的类时，该类中某个函数可能表现出virtual函数的行为，尽管它并没有显式地声明为virtual。当这个类从它的基类继承了一个virtual函数时，这种情况就会发生，并且它还能导致微妙的逻辑错误。通过在整个类的继承层次结构中显式地声明所有的virtual函数，可以避免此类错误。



软件工程知识 13.5

当派生类选择不重写从其基类继承而来的virtual函数时，派生类就会简单地继承它的基类的virtual函数的实现。

如果程序通过指向派生类对象的基类指针调用virtual函数（比如，shapePtr->draw()），程序会根据所指对象的类型而不是指针类型，动态（即执行时）选择正确的派生类draw函数。在执行时（不是编译时）选择合适的调用函数称为动态绑定（dynamic binding）或迟绑定。

当virtual函数通过按名引用特定对象和使用圆点成员选择运算符的方式（例如，squareObject.draw()）被调用时，调用哪个函数在编译时就已经决定（称为静态绑定），所调用的virtual函数正是为该特定对象所属的类定义的（或继承而来的）函数——这并不是多态性行为。因此，用virtual函数进行动态绑定只能通过指针（以及我们即将看到的引用）句柄完成。

现在，让我们看看在我们的雇员类层次结构中virtual函数是怎样导致多态性行为的。图13.8和图13.9分别是类CommissionEmployee和类BasePlusCommissionEmployee的头文件。请注意，它们与图13.1和图13.3中头文件惟一的区别是，我们把每个类中的earnings和print成员函数指定为virtual函数（见图13.8的第30~31行和图13.9的第21~22行）。因为在CommissionEmployee类中earnings和print成员函数是virtual函数，所以BasePlusCommissionEmployee类的earnings和print函数可以重写CommissionEmployee的这两个函数。现在，如果我们将一个基类CommissionEmployee指针指向一个派生类BasePlusCommissionEmployee的对象，并且程序利用这个指针调用earnings或print函数，那么这个BasePlusCommissionEmployee对象中相应的函数就会被调用。而对于类CommissionEmployee和类BasePlusCommissionEmployee的这些成员函数实现来说，都没有任何改动，所以我们可以继续重用它们在图13.2和图13.4中定义的版本。

```
1  // Fig. 13.8: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
```

图 13.8 CommissionEmployee类的头文件将函数earnings和print声明为virtual函数

```
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    virtual double earnings() const; // calculate earnings
31    virtual void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

图 13.8　CommissionEmployee 类的头文件将函数 earnings 和 print 声明为 virtual 函数（续）

```
1  // Fig. 13.9: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

图 13.9　BasePlusCommissionEmployee 类的头文件将函数 earnings 和 print 声明为 virtual 函数

我们修改图13.5中的程序来创建图13.10中的程序。第46~57行再一次演示了一个指向CommissionEmployee对象的CommissionEmployee类指针可以用来调用CommissionEmployee类的功能，一个指向 BasePlusCommissionEmployee 对象的 BasePlusCommissionEmployee 指针可以用来调用BasePlusCommissionEmployee类的功能。第60行使一个基类指针commissionEmployeePtr指向了派生类对象 basePlusCommissionEmployee。请注意，当第 67 行通过这个基类指针调用成员函数 print 时，派生类 BasePlusCommissionEmployee 的 print 成员函数被调用了，所以第 67 行的输出内容和图 13.5 中第 59 行输出的内容不同（那时成员函数没有声明为 virtual 函数）。我们已经看到，把一个成员函数声明为 virtual 函数，将导致程序根据句柄指向的对象的类型，而不是根据句柄的类型，动态地决定要调用的函数版本，决定调用哪个函数，这是多态性的一个例子。再次注意，当指针commissionEmployeePtr指向一个CommissionEmployee对象（第46行）时，调用的是CommissionEmployee类的 print 函数，而当 CommissionEmployeePtr 指向一个 BasePlusCommissionEmployee 对

象时，调用的是BasePlusCommissionEmployee类的print函数。因此，当相同的消息（本例中为print），发送给（通过基类指针）此基类在继承层次中不同的派生类对象时，将会呈现出“多种形式”——这就是多态性的行为。

```
// Fig. 13.10: fig13_10.cpp
// Introducing polymorphism, virtual functions and dynamic binding.
#include <iostream>
using std::cout;
using std::endl;
using std::fixed;

#include <iomanip>
using std::setprecision;

// include class definitions
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
   // create base-class object
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );

   // create base-class pointer
   CommissionEmployee *commissionEmployeePtr = 0;

   // create derived-class object
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

   // create derived-class pointer
   BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

   // output objects using static binding
   cout << "Invoking print function on base-class and derived-class "
      << "\nobjects with static binding\n\n";
   commissionEmployee.print(); // static binding
   cout << "\n\n";
   basePlusCommissionEmployee.print(); // static binding

   // output objects using dynamic binding
   cout << "\n\n\nInvoking print function on base-class and "
      << "derived-class \nobjects with dynamic binding";

   // aim base-class pointer at base-class object and print
   commissionEmployeePtr = &commissionEmployee;
   cout << "\n\nCalling virtual function print with base-class pointer"
      << "\nto base-class object invokes base-class "
      << "print function:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print

   // aim derived-class pointer at derived-class object and print
   basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\nCalling virtual function print with derived-class "
      << "pointer\nto derived-class object invokes derived-class "
      << "print function:\n\n";
```

图 13.10　通过指向派生类对象的基类指针调用派生类的 virtual 函数，以此展示多态性

```
57     basePlusCommissionEmployeePtr->print(); // invokes derived-class print
58
59     // aim base-class pointer at derived-class object and print
60     commissionEmployeePtr = &basePlusCommissionEmployee;
61     cout << "\n\nCalling virtual function print with base-class pointer"
62        << "\nto derived-class object invokes derived-class "
63        << "print function:\n\n";
64
65     // polymorphism; invokes BasePlusCommissionEmployee's print;
66     // base-class pointer to derived-class object
67     commissionEmployeePtr->print();
68     cout << endl;
69     return 0;
70  } // end main
```

```
Invoking print function on base-class and derived-class
objects with static binding

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Invoking print function on base-class and derived-class
objects with dynamic binding

Calling virtual function print with base-class pointer
to base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Calling virtual function print with derived-class pointer
to derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Calling virtual function print with base-class pointer
to derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

图 13.10　通过指向派生类对象的基类指针调用派生类的 virtual 函数，以此展示多态性（续）

某家公司按周支付雇员工资。雇员一共有四类：定薪雇员，不管每周工作多长时间都领取固定的周薪；钟点雇员：按工作的小时数领取工资，并且可以领取超过40个小时之外的加班费；佣金雇员：工资完全是销售业绩提成；带薪佣金雇员，工资是基本工资加销售业绩提成。在这次工资发放阶段，公司决定奖励带薪佣金雇员，把他们的基本工资提高10%。公司想实现一个C++程序多态地执行工资的计算。

我们使用抽象类Employee表示总体意义下的雇员。直接从Employee类派生的是类SalariedEmployee、CommissionEmployee和HourlyEmployee。而BasePlusCommissionEmployee类又是从CommissionEmployee类直接派生的——代表最后一种雇员类型。图13.11中的UML类图显示了多态的雇员工资应用程序中类的继承层次结构。请注意，按照UML惯例，抽象类类名Employee用斜体表示。



图 13.11　雇员类层次结构的 UML 类图

抽象基类Employee声明了类层次结构的“接口”，即，程序可以对所有的Employee类对象调用的一组成员函数集合。每个雇员，不论他或她的工资计算方式如何，都有名字、姓氏以及社会安全号码，因此在抽象基类Employee中含有private数据成员firstName、lastName和socialSecurityNumber。



软件工程知识 13.10

派生类可以从基类继承接口或实现。为“实现继承”（implementation inheritance）而设计的类层次结构往往将功能设置在较高层——每个新派生类继承定义在基类中的一个或多个成员函数，并且派生类使用这些基类定义；而为“接口继承”（interface inheritance）而设计的类层次结构则趋于将功能设置在较低层——基类指定一个或多个应为类继承层次中的每个类定义的函数（即它们有相同的原型），但是各个派生类提供自己对于这些函数的实现。

接下来的几节将实现上述的Employee类层次结构。前五节每节实现一个抽象类或具体类。最后一节构建一个测试程序，它创建所有这些类的对象并多态地处理这些对象。

13.6.1　创建抽象基类：Employee 类

Employee类（见图13.13和图13.14）除了包含操纵Employee类的数据成员的各种获取和设置函数外，还提供成员函数earnings和print。earnings函数当然应用于所有的雇员，但是每项收入的计算取决于雇员的类型。所以我们在基类Employee中把earnings函数声明为纯virtual函数，因为这个函数默认的一种实现是没有任何意义的——没有足够的信息决定应该返回的收入是多少。每个派生类都用合适的实现重写earnings函数。要计算一个雇员的收入，程序把一个雇员对象的地址赋给一个基类Employee指针，然后调用该对象的earings函数。我们维护一个Employee指针的vector对象，每个指针都指向一个Employee对象（当然，不可能有Employee对象，因为Employee是一个抽象类——不过因为继承的关系，Employee的每个派生类的所有对象都可以认为是Employee对象）。程序迭代访问此vector对象并调用每个Employee对象的earnings函数。C++多态地执行这些函数调用。因Employee类中含有纯virtual函数earnings，迫使所有希望成为具体类的那些直接继承Employee类的派生类都重写earnings函数。这使类层次结构的设计人员能够要求每个派生类提供合适的工资计算方式（如果这个派生类的确要成为具体类）。

在Employee类中的print函数显示雇员的名、姓和社会安全号码。我们将看到，每个Employee类的派生类都重写了print函数，输出雇员的类型（例如，“定薪雇员：”）之后，紧接着还输出了雇员的其他信息。

图13.12的图表类继承层次中的五个类，顶部显示的是earnings和print函数名。对于每个类，这张图表显示了每个函数期望的实现结果。请注意，在Employee类中earnings函数被指定为“= 0”，表示它是一个纯virtual函数。每个派生类都重写earnings函数，提供合适的实现。我们没有列出基类Employee的获取和设置函数，因为它们并没有在任何派生类中重写——这些函数都被派生类继承并且“按照原样”使用。

	earnings	print
Employee	= 0	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* (40 * wage) + ((hours - 40) * wage * 1.5)	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage*; hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	baseSalary + (commissionRate * grossSales)	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

图13.12 Employee类层次结构的多态性接口

让我们考虑Employee类的头文件（见图13.13）。public成员函数包括：一个构造函数，该构造函数以名、姓和社会安全号码为参数（第12行）；设置函数，设置了名、姓和社会安全号码（分别是第14、17和20行）；获取函数，返回了名、姓和社会安全号码（分别是第15、18和21行）；纯virtual函数earnings（第24行）和纯virtual函数print（第25行）。

```
1  // Fig. 13.13: Employee.h
2  // Employee abstract base class.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class Employee
10 {
11 public:
12    Employee( const string &, const string &, const string & );
13
14    void setFirstName( const string & ); // set first name
```

图13.13 Employee类的头文件

```
15     string getFirstName() const; // return first name
16
17     void setLastName( const string & ); // set last name
18     string getLastName() const; // return last name
19
20     void setSocialSecurityNumber( const string & ); // set SSN
21     string getSocialSecurityNumber() const; // return SSN
22
23     // pure virtual function makes Employee abstract base class
24     virtual double earnings() const = 0; // pure virtual
25     virtual void print() const; // virtual
26  private:
27     string firstName;
28     string lastName;
29     string socialSecurityNumber;
30  }; // end class Employee
31
32  #endif // EMPLOYEE_H
```

图 13.13　Employee 类的头文件（续）

回想一下我们把 earnings 函数声明为纯 virtual 函数，因为我们首先必须知道具体雇员的类型，才能决定适当的工资计算方法。将这一函数声明为纯 virtual 函数，表明每个具体的派生类必须提供一个适当的 earnings 函数的实现，并且程序可以利用基类 Employee 指针根据雇员的不同类型多态地调用 earnings 函数。

图 13.14 包含了 Employee 类成员函数的实现。没有为纯 virtual 函数 earnings 提供任何实现代码。请注意，Employee 类的构造函数（第 10~15 行）并没有确认社会安全号码的有效性。通常应该提供这类有效性确认。第 12 章的一道习题要求读者确认社会安全号码的有效性，以保证它的格式是 ###-##-#### 的形式，其中每个 # 代表一个数字。

```
1  // Fig. 13.14: Employee.cpp
2  // Abstract-base-class Employee member-function definitions.
3  // Note: No definitions are given for pure virtual functions.
4  #include <iostream>
5  using std::cout;
6
7  #include "Employee.h" // Employee class definition
8
9  // constructor
10 Employee::Employee( const string &first, const string &last,
11    const string &ssn )
12    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14    // empty body
15 } // end Employee constructor
16
17 // set first name
18 void Employee::setFirstName( const string &first )
19 {
20    firstName = first;
21 } // end function setFirstName
22
23 // return first name
24 string Employee::getFirstName() const
25 {
26    return firstName;
27 } // end function getFirstName
28
29 // set last name
30 void Employee::setLastName( const string &last )
31 {
```

图 13.14　Employee 类的实现文件

```
32     lastName = last;
33  } // end function setLastName
34
35  // return last name
36  string Employee::getLastName() const
37  {
38     return lastName;
39  } // end function getLastName
40
41  // set social security number
42  void Employee::setSocialSecurityNumber( const string &ssn )
43  {
44     socialSecurityNumber = ssn; // should validate
45  } // end function setSocialSecurityNumber
46
47  // return social security number
48  string Employee::getSocialSecurityNumber() const
49  {
50     return socialSecurityNumber;
51  } // end function getSocialSecurityNumber
52
53  // print Employee's information (virtual, but not pure virtual)
54  void Employee::print() const
55  {
56     cout << getFirstName() << ' ' << getLastName()
57        << "\nsocial security number: " << getSocialSecurityNumber();
58  } // end function print
```

图 13.14　Employee 类的实现文件（续）

请注意，virtual 函数 print（图 13.14 中的第 54~58 行）提供的实现会在每个派生类中被重写。可是，这些 print 函数都将使用这个抽象类中 print 函数的版本，输出 Employee 类层次结构中所有类共有的信息。

13.6.2　创建具体的派生类：SalariedEmployee 类

SalariedEmployee 类（见图 13.15 和图 13.16）是从 Employee 类（图 13.15 中的第 8 行）派生而来的，其 public 成员函数包括：一个以名、姓、社会安全号码和周薪为参数的构造函数（第 11~12 行）；给数据成员 weeklySalary 赋一个非负值的设置函数（第 14 行）；返回 weeklySalary 值的获取函数（第 15 行）；计算一个 Salaried Employee 雇员收入的 virtual 函数 earnings（第 18 行）；打印雇员信息的 virtual 函数 print（第 19 行），它输出的内容依次是雇员类型（即“salaried employee:”），以及由基类 Employee 的 print 函数和 SalariedEmployee 类的 getWeeklySalary 函数产生的雇员的特定信息（第 19 行）。

图 13.16 包含了 SalariedEmployee 类的成员函数的实现。类的构造函数把名、姓和社会安全号码传递给基类 Employee 的构造函数（第 11 行），从而对从基类继承但在派生类中不可访问的 private 数据成员进行了初始化。Earnings 函数（第 30~33 行）重写了基类 Emloyee 中的纯 virtual 函数 earnings，提供了返回 SalariedEmployee 雇员周薪的具体实现。如果我们不具体实现 earnings，SalariedEmployee 类会成为一个抽象类，那么所有试图实例化该类对象的操作都会导致编译错误（当然，这里我们希望 SalariedEmployee 类是一个具体类）。请注意，在 SalariedEmployee 类的头文件中，我们将成员函数 earnings 和 print 声明为 virtual 函数（图 13.15 中的第 18~19 行）——实际上，在这两个函数的前面添加 virtual 关键字是多余的。在基类 Employee 中已经定义它们为 virtual 函数，所以它们在整个类继承层次中都将保持为 virtual 函数。回忆在“良好的编程习惯 13.1”中我们告诉读者，为了使程序更加清晰可读，最好在类层次结构的每一级中都把这样的函数显式地声明为 virtual 函数。

```
1  // Fig. 13.15: SalariedEmployee.h
2  // SalariedEmployee class derived from Employee.
3  #ifndef SALARIED_H
4  #define SALARIED_H
5
6  #include "Employee.h" // Employee class definition
7
8  class SalariedEmployee : public Employee
9  {
10 public:
11    SalariedEmployee( const string &, const string &,
12       const string &, double = 0.0 );
13
14    void setWeeklySalary( double ); // set weekly salary
15    double getWeeklySalary() const; // return weekly salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; // print SalariedEmployee object
20 private:
21    double weeklySalary; // salary per week
22 }; // end class SalariedEmployee
23
24 #endif // SALARIED_H
```

图 13.15　SalariedEmployee 类的头文件

```
1  // Fig. 13.16: SalariedEmployee.cpp
2  // SalariedEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "SalariedEmployee.h" // SalariedEmployee class definition
7
8  // constructor
9  SalariedEmployee::SalariedEmployee( const string &first,
10    const string &last, const string &ssn, double salary )
11    : Employee( first, last, ssn )
12 {
13    setWeeklySalary( salary );
14 } // end SalariedEmployee constructor
15
16 // set salary
17 void SalariedEmployee::setWeeklySalary( double salary )
18 {
19    weeklySalary = ( salary < 0.0 ) ? 0.0 : salary;
20 } // end function setWeeklySalary
21
22 // return salary
23 double SalariedEmployee::getWeeklySalary() const
24 {
25    return weeklySalary;
26 } // end function getWeeklySalary
27
28 // calculate earnings;
29 // override pure virtual function earnings in Employee
30 double SalariedEmployee::earnings() const
31 {
32    return getWeeklySalary();
33 } // end function earnings
```

图 13.16　SalariedEmployee 类的实现文件

```
34
35 // print SalariedEmployee's information
36 void SalariedEmployee::print() const
37 {
38    cout << "salaried employee: ";
39    Employee::print(); // reuse abstract base-class print function
40    cout << "\nweekly salary: " << getWeeklySalary();
41 } // end function print
```

图 13.16 SalariedEmployee 类的实现文件（续）

SalariedEmployee 类的 print 函数（图 13.16 中的第 36~41 行）重写了基类 Employee 的 print 函数。如果 SalariedEmployee 类不重写 print 函数，SalariedEmployee 类将继承基类 Employee 中的 print 函数。那样，SalariedEmployee 类的 print 函数只是简单地返回雇员的全名和社会安全号码，而这些信息并不能充分地描述一个 SalariedEmployee 雇员。为了打印 SalariedEmployee 雇员的完整信息，派生类的 print 函数首先输出 "salaried employee: "，然后通过使用二元作用域分辨运算符（第 39 行）调用基类的 print 函数输出基类 Employee 的信息（即名、姓和社会安全号码）——这是一个很好的代码重用的例子。SalariedEmployee 类的 print 函数的输出还包括通过调用 getWeeklySalary 函数获得的该雇员的周薪。

13.6.3 创建具体的派生类：HourlyEmployee 类

HourlyEmployee 类（见图 13.17 和图 13.18）也是 Employee 类（图 13.17 中的第 8 行）的派生类。其 public 成员函数包括：一个以名、姓、社会安全号码、每小时工资和工作小时数为参数的构造函数（第 11~12 行），给数据成员 wage 和 hours 赋新值的设置函数（第 14 和 17 行），返回 wage 和 hours 值的获取函数（第 15 和 18 行），计算一个 HourlyEmployee 雇员收入的 virtual 函数 earnings（第 21 行），输出雇员的类型（即 "hourly employee:"），以及雇员的详细信息的 virtual 函数 print（第 22 行）。

```
1  // Fig. 13.17: HourlyEmployee.h
2  // HourlyEmployee class definition.
3  #ifndef HOURLY_H
4  #define HOURLY_H
5
6  #include "Employee.h" // Employee class definition
7
8  class HourlyEmployee : public Employee
9  {
10 public:
11    HourlyEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0 );
13
14    void setWage( double ); // set hourly wage
15    double getWage() const; // return hourly wage
16
17    void setHours( double ); // set hours worked
18    double getHours() const; // return hours worked
19
20    // keyword virtual signals intent to override
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print HourlyEmployee object
23 private:
24    double wage; // wage per hour
25    double hours; // hours worked for week
26 }; // end class HourlyEmployee
27
28 #endif // HOURLY_H
```

图 13.17 HourlyEmployee 类的头文件

```
1  // Fig. 13.18: HourlyEmployee.cpp
2  // HourlyEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "HourlyEmployee.h" // HourlyEmployee class definition
7
8  // constructor
9  HourlyEmployee::HourlyEmployee( const string &first, const string &last,
10    const string &ssn, double hourlyWage, double hoursWorked )
11    : Employee( first, last, ssn )
12 {
13    setWage( hourlyWage ); // validate hourly wage
14    setHours( hoursWorked ); // validate hours worked
15 } // end HourlyEmployee constructor
16
17 // set wage
18 void HourlyEmployee::setWage( double hourlyWage )
19 {
20    wage = ( hourlyWage < 0.0 ? 0.0 : hourlyWage );
21 } // end function setWage
22
23 // return wage
24 double HourlyEmployee::getWage() const
25 {
26    return wage;
27 } // end function getWage
28
29 // set hours worked
30 void HourlyEmployee::setHours( double hoursWorked )
31 {
32    hours = ( ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
33       hoursWorked : 0.0 );
34 } // end function setHours
35
36 // return hours worked
37 double HourlyEmployee::getHours() const
38 {
39    return hours;
40 } // end function getHours
41
42 // calculate earnings;
43 // override pure virtual function earnings in Employee
44 double HourlyEmployee::earnings() const
45 {
46    if ( getHours() <= 40 ) // no overtime
47       return getWage() * getHours();
48    else
49       return 40 * getWage() + ( ( getHours() - 40 ) * getWage() * 1.5 );
50 } // end function earnings
51
52 // print HourlyEmployee's information
53 void HourlyEmployee::print() const
54 {
55    cout << "hourly employee: ";
56    Employee::print(); // code reuse
57    cout << "\nhourly wage: " << getWage() <<
58       "; hours worked: " << getHours();
59 } // end function print
```

图 13.18　HourlyEmployee 类的实现文件

图 13.18 包含 HourlyEmployee 类成员函数的实现。第 18~21 行和第 30~34 行定义的设置函数分别给数据成员 wage 和 hours 赋新值。setWage 函数（第 18~21 行）保证 wage 是非负的，setHours 函数（第 30~34 行）保证数据成员 hours 的值在 0 和 168（一周总共的小时数）之间。HourlyEmployee 类的获取函数的实现在第 24~27 行和第 37~40 行。我们没有把这些函数声明为 virtual，所以从 HourlyEmployee 类派生的类不能重写它们（尽管派生类可以重新定义它们）。请注意 HourlyEmployee 类的构造函数，它和 SalariedEmployee 类的构造函数一样，把名、姓和社会安全号码传递给基类 Employee 的构造函数（第 11 行），从而对继承于基类的 private 数据成员进行了初始化。另外，HourlyEmployee 类的 print 函数调用基类的 print 函数（第 56 行）来输出 Employee 特定的信息（也就是名、姓和社会安全号码） —— 这又是一个很好的代码重用的例子。

13.6.4 创建具体的派生类：CommissionEmployee 类

CommissionEmployee 类（见图 13.19 和图 13.20）同样也是 Employee 类（图 13.19 中的第 8 行）的派生类。其 public 成员函数的实现（见图 13.20）包括：一个以名、姓、社会安全号码、销售总额和提成比例为参数的构造函数（第 9~15 行）；给数据成员 commissionRate 和 grossSales 分别赋予新值的设置函数（第 18~21 行和第 30~33 行）；返回数据成员 commissionRate 和 grossSales 值的获取函数（第 24~27 行和第 36~39 行）；计算一个 CommissionEmployee 收入的 earnings 函数（第 43~46 行）；输出雇员类型（即 “commission employee:” 以及雇员的详细信息）的 print 函数（第 49~55 行）。CommissionEmployee 类的构造函数依然是把名、姓和社会安全号码传递给基类 Employee 的构造函数（第 11 行），从而初始化从基类继承而来的 private 数据成员。print 函数调用基类的 print 函数（第 52 行）来显示 Employee 特定的信息（也就是名、姓和社会安全号码）。

```
1  // Fig. 13.19: CommissionEmployee.h
2  // CommissionEmployee class derived from Employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include "Employee.h" // Employee class definition
7
8  class CommissionEmployee : public Employee
9  {
10 public:
11    CommissionEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0 );
13
14    void setCommissionRate( double ); // set commission rate
15    double getCommissionRate() const; // return commission rate
16
17    void setGrossSales( double ); // set gross sales amount
18    double getGrossSales() const; // return gross sales amount
19
20    // keyword virtual signals intent to override
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print CommissionEmployee object
23 private:
24    double grossSales; // gross weekly sales
25    double commissionRate; // commission percentage
26 }; // end class CommissionEmployee
27
28 #endif // COMMISSION_H
```

图 13.19 CommissionEmployee 类的头文件

```
1  // Fig. 13.20: CommissionEmployee.cpp
2  // CommissionEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  // constructor
9  CommissionEmployee::CommissionEmployee( const string &first,
10    const string &last, const string &ssn, double sales, double rate )
11    : Employee( first, last, ssn )
12 {
13    setGrossSales( sales );
14    setCommissionRate( rate );
15 } // end CommissionEmployee constructor
16
17 // set commission rate
18 void CommissionEmployee::setCommissionRate( double rate )
19 {
20    commissionRate = ( ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0 );
21 } // end function setCommissionRate
22
23 // return commission rate
24 double CommissionEmployee::getCommissionRate() const
25 {
26     return commissionRate;
27 } // end function getCommissionRate
28
29 // set gross sales amount
30 void CommissionEmployee::setGrossSales( double sales )
31 {
32    grossSales = ( ( sales < 0.0 ) ? 0.0 : sales );
33 } // end function setGrossSales
34
35 // return gross sales amount
36 double CommissionEmployee::getGrossSales() const
37 {
38     return grossSales;
39 } // end function getGrossSales
40
41 // calculate earnings;
42 // override pure virtual function earnings in Employee
43 double CommissionEmployee::earnings() const
44 {
45    return getCommissionRate() * getGrossSales();
46 } // end function earnings
47
48 // print CommissionEmployee's information
49 void CommissionEmployee::print() const
50 {
51    cout << "commission employee: ";
52    Employee::print(); // code reuse
53    cout << "\ngross sales: " << getGrossSales()
54       << "; commission rate: " << getCommissionRate();
55 } // end function print
```

图 13.20　CommissionEmployee 类的实现文件

13.6.5　创建间接派生的具体类：BasePlusCommissionEmployee 类

BasePlusCommissionEmployee 类（见图 13.21 和图 13.22）是 CommissionEmployee 类（见图 13.21 中的第 8 行）的直接派生类，因此是 Employee 类的间接派生类。BasePlusCommissionEmployee 类的成员函数实现包括了一个以名、姓、社会安全号码、销售总额、提成比例和基本工资为参数的构造函数（见图 13.22 中的第 10~16 行）。然后，该构造函数把名、姓、社会安全号码、销售总额、提

成比例传递给 CommissionEmployee 类的构造函数（第 13 行），从而初始化继承的成员。BasePlusCommissionEmployee类也包含了一个设置函数（第19~22行）（用于给数据成员 baseSalary 赋新值）和一个 get 函数（第 25~28 行）（用于返回 baseSalary 的值）。earnings 函数（第 32~35 行）计算一个 BasePlusCommissionEmployee 雇员的收入。请注意，函数 earnings 中的第 34 行调用基类 CommissionEmployee 的 earnings 函数以计算该雇员收入中提成所得的部分。这是一个很好的代码重用的例子。BasePlusCommissionEmployee 类的 print 函数（第 38~43 行）首先输出 "base-salaried"，然后是它的基类 CommissionEmployee 的 print 函数的输出（又是一个代码重用的例子），最后输出基本工资。输出结果以 "base-salaried commission employee:" 开始，接着是 BasePlusCommissionEmployee 对象的其余信息。回忆一下 CommissionEmployee 类的 print 函数，它通过调用其基类（即 Emloyee 类）的 print 函数显示了雇员的名、姓和社会安全号码——仍是一个代码重用的例子。请注意 BasePlusCommissionEmployee类的print函数引发了跨越Employee类的三级类层次的一连串的函数调用。

```
1  // Fig. 13.21: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from Employee.
3  #ifndef BASEPLUS_H
4  #define BASEPLUS_H
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  class BasePlusCommissionEmployee : public CommissionEmployee
9  {
10 public:
11    BasePlusCommissionEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0, double = 0.0 );
13
14    void setBaseSalary( double ); // set base salary
15    double getBaseSalary() const; // return base salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; // print BasePlusCommissionEmployee object
20 private:
21    double baseSalary; // base salary per week
22 }; // end class BasePlusCommissionEmployee
23
24 #endif // BASEPLUS_H
```

图 13.21　BasePlusCommissionEmployee 类的头文件

```
1  // Fig. 13.22: BasePlusCommissionEmployee.cpp
2  // BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string &first, const string &last, const string &ssn,
12    double sales, double rate, double salary )
13    : CommissionEmployee( first, last, ssn, sales, rate )
14 {
15    setBaseSalary( salary ); // validate and store base salary
16 } // end BasePlusCommissionEmployee constructor
17
18 // set base salary
19 void BasePlusCommissionEmployee::setBaseSalary( double salary )
20 {
```

图 13.22　BasePlusCommissionEmployee 类的实现文件

```
21    baseSalary = ( ( salary < 0.0 ) ? 0.0 : salary );
22 } // end function setBaseSalary
23
24 // return base salary
25 double BasePlusCommissionEmployee::getBaseSalary() const
26 {
27    return baseSalary;
28 } // end function getBaseSalary
29
30 // calculate earnings;
31 // override pure virtual function earnings in Employee
32 double BasePlusCommissionEmployee::earnings() const
33 {
34    return getBaseSalary() + CommissionEmployee::earnings();
35 } // end function earnings
36
37 // print BasePlusCommissionEmployee's information
38 void BasePlusCommissionEmployee::print() const
39 {
40    cout << "base-salaried ";
41    CommissionEmployee::print(); // code reuse
42    cout << "; base salary: " << getBaseSalary();
43 } // end function print
```

图 13.22　BasePlusCommissionEmployee 类的实现文件（续）

13.6.6　演示多态性的执行过程

为了测试Employee类的层次结构，图 13.23 中的程序为4个具体类 SalariedEmployee、Hourly-Employee、CommissionEmployee 和 BasePlusCommissionEmployee 的每一个都创建了一个对象。程序首先用静态绑定方式对这些对象进行了操作，然后使用 Employee 指针的 vector 对象多态地对这些对象进行了操作。第 31~38 行创建了这 4 个 Employee 具体派生类的各自对象。第 43~51 行输出每个雇员对象的信息和收入。第 43~51 行上的每个成员函数的调用都是一个（在编译时）静态绑定的实例，因为我们使用的是名称句柄（不是可以在执行时设置的指针或引用），编译器可以识别每个对象的类型，从而决定应该调用哪个 print 和 earnings 函数。

```
1  // Fig. 13.23: fig13_23.cpp
2  // Processing Employee derived-class objects individually
3  // and polymorphically using dynamic binding.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 #include <vector>
13 using std::vector;
14
15 // include definitions of classes in Employee hierarchy
16 #include "Employee.h"
17 #include "SalariedEmployee.h"
18 #include "HourlyEmployee.h"
19 #include "CommissionEmployee.h"
20 #include "BasePlusCommissionEmployee.h"
21
22 void virtualViaPointer( const Employee * const ); // prototype
23 void virtualViaReference( const Employee & ); // prototype
24
25 int main()
26 {
27    // set floating-point output formatting
```

图 13.23　Employee 类层次结构的驱动程序

```
28    cout << fixed << setprecision( 2 );
29
30    // create derived-class objects
31    SalariedEmployee salariedEmployee(
32       "John", "Smith", "111-11-1111", 800 );
33    HourlyEmployee hourlyEmployee(
34       "Karen", "Price", "222-22-2222", 16.75, 40 );
35    CommissionEmployee commissionEmployee(
36       "Sue", "Jones", "333-33-3333", 10000, .06 );
37    BasePlusCommissionEmployee basePlusCommissionEmployee(
38       "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
39
40    cout << "Employees processed individually using static binding:\n\n";
41
42    // output each Employee's information and earnings using static binding
43    salariedEmployee.print();
44    cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
45    hourlyEmployee.print();
46    cout << "\nearned $" << hourlyEmployee.earnings() << "\n\n";
47    commissionEmployee.print();
48    cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
49    basePlusCommissionEmployee.print();
50    cout << "\nearned $" << basePlusCommissionEmployee.earnings()
51       << "\n\n";
52
53    // create vector of four base-class pointers
54    vector < Employee * > employees( 4 );
55
56    // initialize vector with Employees
57    employees[ 0 ] = &salariedEmployee;
58    employees[ 1 ] = &hourlyEmployee;
59    employees[ 2 ] = &commissionEmployee;
60    employees[ 3 ] = &basePlusCommissionEmployee;
61
62    cout << "Employees processed polymorphically via dynamic binding:\n\n";
63
64    // call virtualViaPointer to print each Employee's information
65    // and earnings using dynamic binding
66    cout << "Virtual function calls made off base-class pointers:\n\n";
67
68    for ( size_t i = 0; i < employees.size(); i++ )
69       virtualViaPointer( employees[ i ] );
70
71    // call virtualViaReference to print each Employee's information
72    // and earnings using dynamic binding
73    cout << "Virtual function calls made off base-class references:\n\n";
74
75    for ( size_t i = 0; i < employees.size(); i++ )
76       virtualViaReference( *employees[ i ] ); // note dereferencing
77
78    return 0;
79 } // end main
80
81 // call Employee virtual functions print and earnings off a
82 // base-class pointer using dynamic binding
83 void virtualViaPointer( const Employee * const baseClassPtr )
84 {
85    baseClassPtr->print();
86    cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
87 } // end function virtualViaPointer
88
89 // call Employee virtual functions print and earnings off a
90 // base-class reference using dynamic binding
91 void virtualViaReference( const Employee &baseClassRef )
92 {
```

图 13.23 Employee 类层次结构的驱动程序（续）

```
93       baseClassRef.print();
94       cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
95    } // end function virtualViaReference
```

```
Employees processed individually using static binding:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00

Employees processed polymorphically using dynamic binding:

Virtual function calls made off base-class pointers:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00

Virtual function calls made off base-class references:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

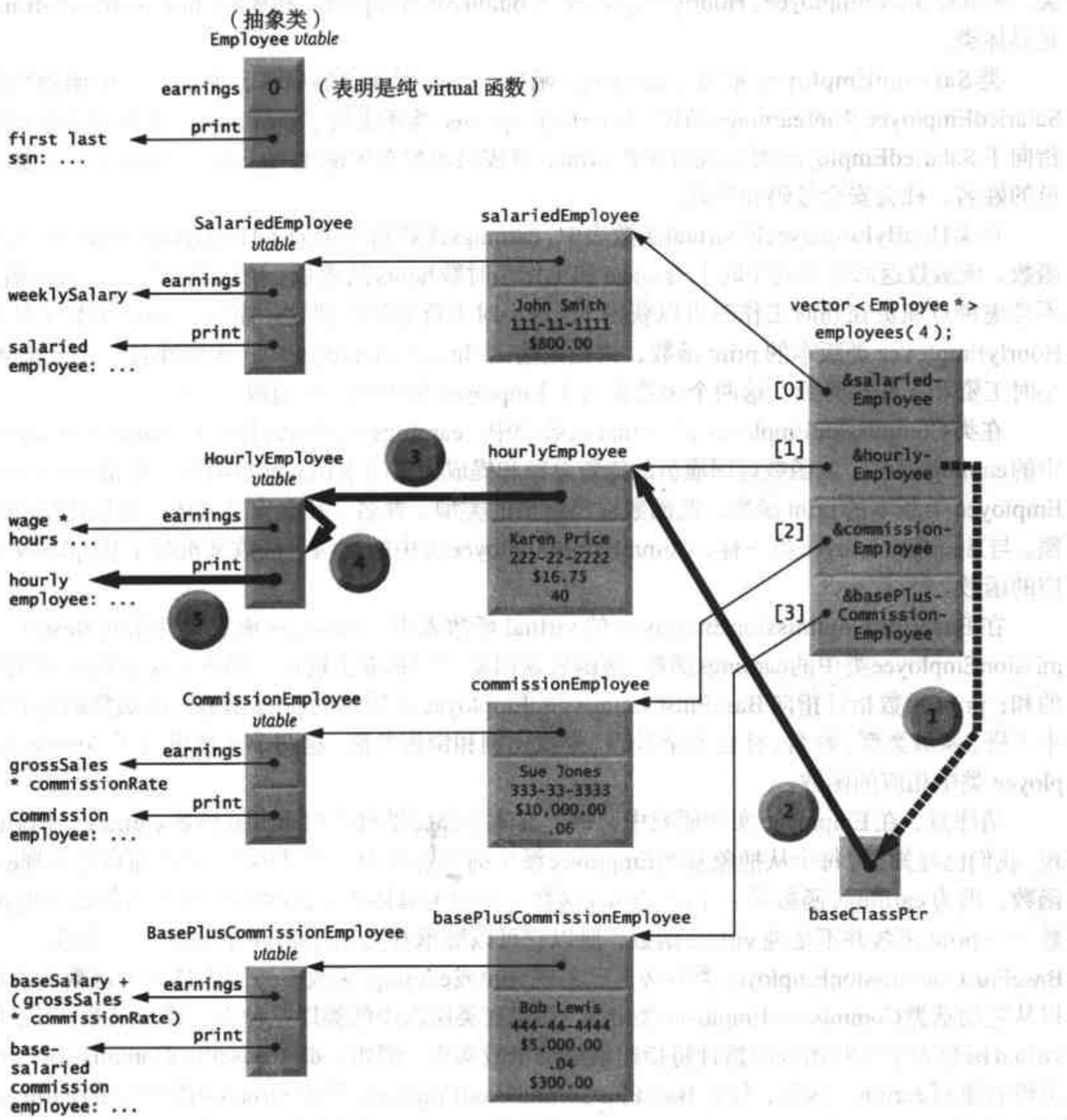
hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

图 13.23　Employee 类层次结构的驱动程序（续）

当 C++ 编译含有一个或多个 virtual 函数的类时，它为这个类创建一个 virtual 函数表（virtual function table，简称 vtable）。每次调用该类的 virtual 函数时，运行程序都会利用 vtable 选择正确的函数实现。图 13.24 的最左列展示了类 Employee、SalariedEmployee、HourlyEmployee、CommissionEmployee 和 BasePlusCommissionEmployee 的 virtual 函数表。



当 baseClassPtr 指向对象 hourlyEmployee 时，virtual 函数调用 baseClassPtr->print()的控制流：

1. 将 &hourlyEmployee 传递给 baseClassPtr
2. 到达对象 hourlyEmployee
3. 到达对象 HourlyEmployee
4. 到达 vtable 中 print 指针
5. 执行 HourlyEmployee 的 print 函数

图 13.24　virtual 函数调用的工作原理

这一节演示了运行时类型信息（RTTI）和动态强制类型转换的强大功能，它们使程序在运行时能够判定对象的类型，从而对对象进行操作。

有些编译器，像Microsoft Visual C++ .NET，需要激活RTTI功能，使之可以在程序中使用。查阅你所用编译器的文档，确定你的编译器是否有类似的要求。在Visual C++ .NET中要为当前工程激活RTTI功能，应首先选择Project菜单，接着选择properties option选项。在出现的Property Pages对话框中，选择Configuration Properties > C/C++ > Language，然后在Enable Run-Time Type Info旁边的组合框中选择Yes（/GR）。最后，点击OK按钮保存设置。

图13.25中的程序使用13.6节开发的Employee类层次结构，并给每个BasePlusCommissionEmployee类雇员增加10%的基本工资。第31行声明了一个含有4个元素的vector对象employees，存储指向Employee对象的指针。第34~41行将动态分配的类SalariedEmployee（见图13.15和图13.16）、HourlyEmployee（见图13.17和图13.18）、CommissionEmployee（见图13.19和图13.20）和BasePlusCommissionEmployee（见图13.21和图13.22）的4个对象地址分别赋给了employees的4个元素。

```
1  // Fig. 13.25: fig13_25.cpp
2  // Demonstrating downcasting and run-time type information.
3  // NOTE: For this example to run in Visual C++ .NET,
4  // you need to enable RTTI (Run-Time Type Info) for the project.
5  #include <iostream>
6  using std::cout;
7  using std::endl;
8  using std::fixed;
9
10 #include <iomanip>
11 using std::setprecision;
12
13 #include <vector>
14 using std::vector;
15
16 #include <typeinfo>
17
18 // include definitions of classes in Employee hierarchy
19 #include "Employee.h"
20 #include "SalariedEmployee.h"
21 #include "HourlyEmployee.h"
22 #include "CommissionEmployee.h"
23 #include "BasePlusCommissionEmployee.h"
24
25 int main()
26 {
27    // set floating-point output formatting
28    cout << fixed << setprecision( 2 );
29
30    // create vector of four base-class pointers
31    vector < Employee * > employees( 4 );
32
33    // initialize vector with various kinds of Employees
34    employees[ 0 ] = new SalariedEmployee(
35       "John", "Smith", "111-11-1111", 800 );
36    employees[ 1 ] = new HourlyEmployee(
37       "Karen", "Price", "222-22-2222", 16.75, 40 );
38    employees[ 2 ] = new CommissionEmployee(
39       "Sue", "Jones", "333-33-3333", 10000, .06 );
40    employees[ 3 ] = new BasePlusCommissionEmployee(
41       "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
42
43    // polymorphically process each element in vector employees
```

图13.25 演示向下强制类型转换和运行时类型信息

```
   for ( size_t i = 0; i < employees.size(); i++ )
   {
      employees[ i ]->print(); // output employee information
      cout << endl;

      // downcast pointer
      BasePlusCommissionEmployee *derivedPtr =
         dynamic_cast < BasePlusCommissionEmployee * >
            ( employees[ i ] );

      // determine whether element points to base-salaried
      // commission employee
      if ( derivedPtr != 0 ) // 0 if not a BasePlusCommissionEmployee
      {
         double oldBaseSalary = derivedPtr->getBaseSalary();
         cout << "old base salary: $" << oldBaseSalary << endl;
         derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
         cout << "new base salary with 10% increase is: $"
            << derivedPtr->getBaseSalary() << endl;
      } // end if

      cout << "earned $" << employees[ i ]->earnings() << "\n\n";
   } // end for

   // release objects pointed to by vector's elements
   for ( size_t j = 0; j < employees.size(); j++ )
   {
      // output class name
      cout << "deleting object of "
         << typeid( *employees[ j ] ).name() << endl;

      delete employees[ j ];
   } // end for

   return 0;
} // end main
```

```
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

deleting object of class SalariedEmployee
deleting object of class HourlyEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
```

图 13.25　演示向下强制类型转换和运行时类型信息（续）

附录B　ASCII字符集



ASCII字符集如图B.1所示。

ASCII字符集

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	lf	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	\|	}	~	del		

图B.1　ASCII字符集

该图左侧的数字是与字符码对应的十进制值（0~127）中的高位数字，而该图顶部的数字是字符码的低位数字。例如，“F”的字符码是70，“&”的字符码是38。

F.1 简介

在第2章，我们知道了错误的类型有两种——编译错误和逻辑错误，并学习了怎样从代码中消除编译错误。逻辑错误也称为程序缺陷（bug），并不会使程序编译失败，但是在程序运行时，它的确使程序产生错误的结果。绝大多数的C++编译器厂商都提供称为调试器（debugger）的软件，使你可以监控程序的执行，以便定位和消除逻辑错误。调试器将成为你最为重要的程序开发工具之一。本附录展示Visual Studio.NET调试器的主要特性。附录G讨论GNU C++调试器的特性和功能。我们提供了一些免费的Dive Into系列出版物，帮助学生和教师熟悉随各种各样其他开发工具而提供的调试器。这些出版物在随书附赠的光盘中提供，也可以从www.deitel.com/books/downloads下载。

F.2 断点和Continue命令

我们就从研究断点开始进行调试器的学习，所谓断点就是指可以在任何可执行的代码行上设置的记号。当程序运行遇到断点时，执行就会暂停，允许你检查其中变量的值以帮助判定是否存在逻辑错误。例如，你可以检查存放计算结果的变量值来判断计算是否正确。注意，在函数的不可执行的代码行（譬如注释行）上设置断点，事实上是将该断点设置标记在下一条可执行的代码行上。

为了举例说明调试器的特性，我们使用图F.3所列的程序，该程序创建和操作类Account（见图F.1和图F.2）的一个对象。执行开始于main函数（见图F.3中的第12~30行）。第14行创建了一个Account的对象，拥有初始的余额50.00美元。Account的构造函数（见图F.2中的第10~22行）接受一个参数，指定Account对象数据成员balance的初始值。图F.3的第17行使用Account的成员函数getBalance输出初始账户余额。第19行声明了一个局部变量withdrawalAmount，用于存储用户要提取的金额。第21行提示用户输入提取的金额，第22行将此金额输入到withdrawalAmount。第25行使用成员函数debit从这个Account对象的balance中减去提取金额。最后，第28行显示新的balance值。

```
1  // Fig. F.1: Account.h
2  // Definition of Account class.
3
4  class Account
5  {
6  public:
7     Account( int ); // constructor initializes balance
8     void credit( int ); // add an amount to the account balance
9     void debit( int ); // subtract an amount from the account balance
10    int getBalance(); // return the account balance
11 private:
12    int balance; // data member that stores the balance
13 }; // end class Account
```

图F.1 类Account的头文件

```
1  // Fig. F.2: Account.cpp
2  // Member-function definitions for class Account.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Account.h" // include definition of class Account
8
9  // Account constructor initializes data member balance
10 Account::Account( int initialBalance )
11 {
12    balance = 0; // assume that the balance begins at 0
```

图F.2 类Account的定义

```
13
14    // if initialBalance is greater than 0, set this value as the
15    // balance of the Account; otherwise, balance remains 0
16    if ( initialBalance > 0 )
17       balance = initialBalance;
18
19    // if initialBalance is negative, print error message
20    if ( initialBalance < 0 )
21       cout << "Error: Initial balance cannot be negative.\n" << endl;
22 } // end Account constructor
23
24 // credit (add) an amount to the account balance
25 void Account::credit( int amount )
26 {
27    balance = balance + amount; // add amount to balance
28 } // end function credit
29
30 // debit (subtract) an amount from the account balance
31 void Account::debit( int amount )
32 {
33    if ( amount <= balance ) // debit amount does not exceed balance
34       balance = balance - amount;
35
36    else // debit amount exceeds balance
37       cout << "Debit amount exceeded account balance.\n" << endl;
38 } // end function debit
39
40 // return the account balance
41 int Account::getBalance()
42 {
43    return balance; // gives the value of balance to the calling function
44 } // end function getBalance
```

图 F.2　类 Account 的定义（续）

```
1  // Fig. F.3: figF_03.cpp
2  // Create and manipulate Account objects.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  // include definition of class Account from Account.h
9  #include "Account.h"
10
11 // function main begins program execution
12 int main()
13 {
14    Account account1( 50 ); // create Account object
15
16    // display initial balance of each object
17    cout << "account1 balance: $" << account1.getBalance() << endl;
18
19    int withdrawalAmount; // stores withdrawal amount read from user
20
21    cout << "\nEnter withdrawal amount for account1: "; // prompt
22    cin >> withdrawalAmount; // obtain user input
23    cout << "\nattempting to subtract " << withdrawalAmount
24       << " from account1 balance\n\n";
25    account1.debit( withdrawalAmount ); // try to subtract from account1
26
27    // display balances
28    cout << "account1 balance: $" << account1.getBalance() << endl;
29    return 0; // indicate successful termination
30 } // end main
```

图 F.3　为了调试而测试类

在接下来的步骤中，将使用断点和各种调试命令来检查在图 F.3 中声明的变量 withdrawalAmount 的值。

1. **激活调试器。**调试器默认时是激活的。如果没有激活，必须修改工具条上的解决方法配置组合框的设定（见图F.4）。为此，点击该组合框的下拉箭头，看到解决方法配置组合框，然后选择 Debug。这样，工具条上会在解决方法配置组合框中显示 Debug。

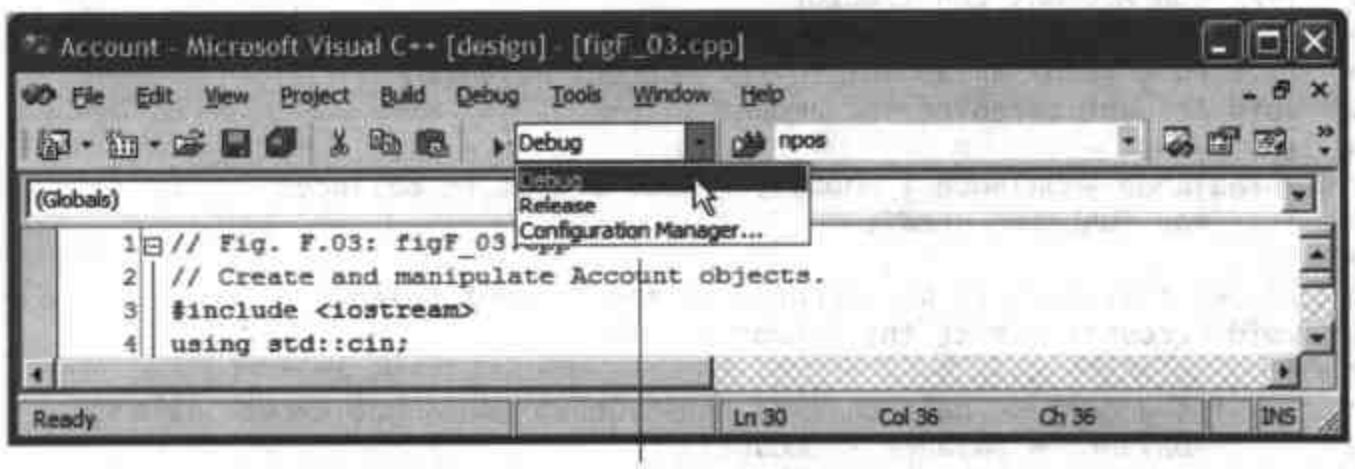


图 F.4 激活调试器

2. **在 Visual Studio .NET 中插入断点。**为了在 Visual Studio.NET 中插入断点，或者在想插入断点的代码行旁边的空白指示条（图F.5中代码窗口左边的页边灰白处）内点击，或者在代码行上点击右键并选择Insert Breakpoint。根据需要，可以插入任意多个断点。在图F.5中代码的第21~25行设置断点。在点击的空白指示条上显示的实心栗色圆圈表明了已经设置了断点（见图F.5）。当程序运行时，调试器在每个标有断点的行处停止执行程序。当调试器暂停程序的执行时，我们说程序处于中断模式。断点可以在程序运行前设置，也可以在程序运行过程中设置。

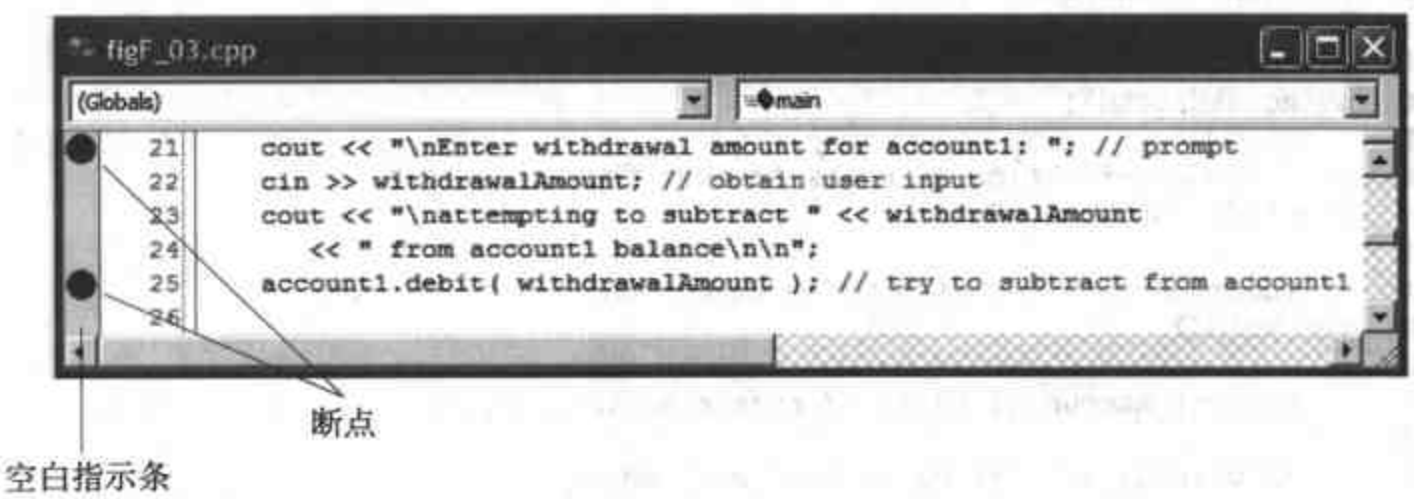


图 F.5 设置两个断点

3. **开始调试过程。**当在代码编辑器中设置完断点后，选择Build➪Build Solution来编译程序，接着选择Debug➪Start开始调试过程。在C++程序的调试期间，会出现一个Command Prompt窗口（见图F.6），允许程序的交互（输入和输出）。程序遇到第21行的断点时就会暂停。此时，该IDE（集成开发环境）的标题栏将显示[break]（见图F.7），表示IDE正处于中断模式。



图 F.6 Inventory 程序运行

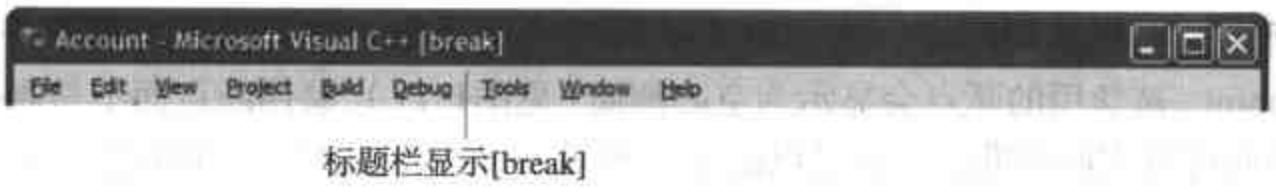


图 F.7　该 IDE 的标题栏显示[break]

4. **检查程序执行。**程序的执行暂停在第一个断点处（第21行），IDE变成了活动窗口（见图F.8）。第21行左边的箭头指出该行包含下一条要执行的语句。注意，我们在图像上增加了亮条。你的代码上不会有这种亮条。

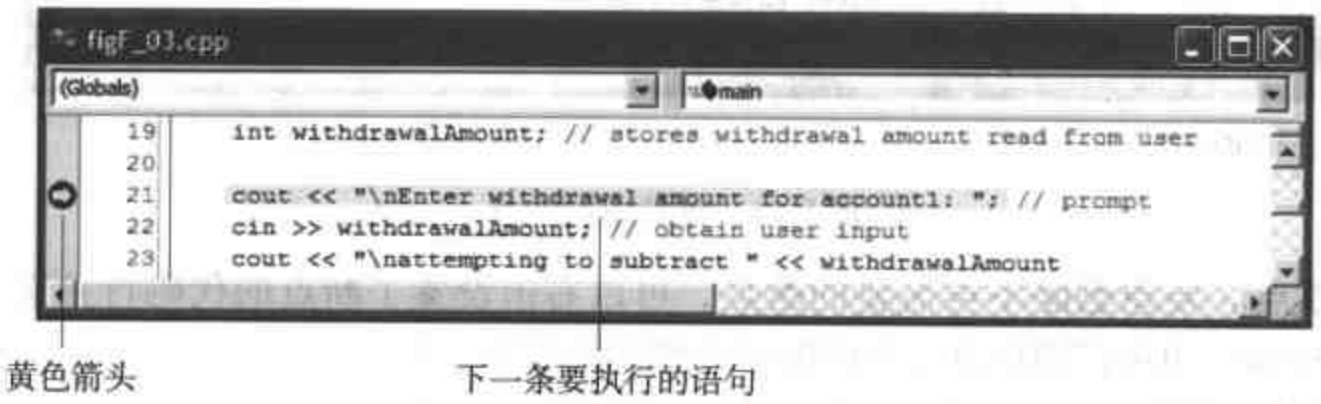


图 F.8　在第一个断点处暂停执行程序

5. **使用 Continue 命令恢复执行。**要想恢复程序的执行，选择Debug➪Continue。Continue命令将执行下一条可执行语句和下一个断点或是main结尾（当然应是先遇到的一个）之间的所有语句。程序继续执行，然后在第22行暂停等待输入。输入13作为提取金额。程序接着执行直到在下一个断点处暂停，这里断点在第25行。注意，当把鼠标放在变量名 withdrawal-Amount 上时，在快速信息框上（见图F.9）将显示该变量存储的值。从某种意义上说，你正在计算机内部查看一个变量的值。正如所见，这可以帮你发现程序中的逻辑错误。

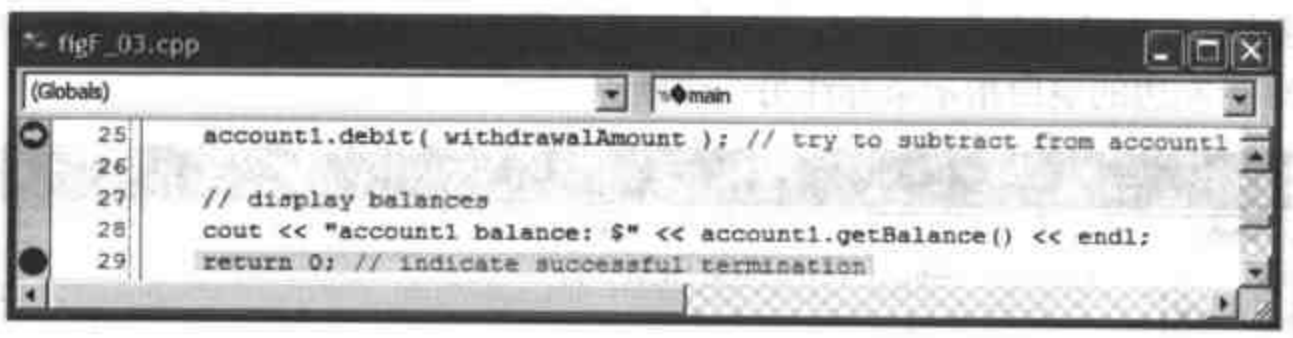


图 F.9　在第 29 行处设置一个断点

6. **在 return 语句上设置断点。**点击源代码第29行左边的空白指示条设置断点（见图F.9）。这样可以防止程序在显示结果之后太快关闭程序。当不再有断点来暂停程序时，程序将直接执行完毕，并关闭Command Prompt窗口。如果不设置这个断点，将无法在控制台窗口关闭前看到程序的输出。
7. **继续程序的执行。**使用 Debug➪Continue 命令执行第25行。程序显示其计算的结果（见图F.10）。



图 F.10　程序输出

8. **禁用断点。**要想禁用断点，右击设置了断点的代码行（或是断点本身）并选择 Disable Breakpoint。被禁用的断点会显示为空心圆圈（见图F.11）。禁用断点而不是删除它是为了在程序中允许再次激活断点（通过点击空心圆圈）。也可以通过右击标记了空心圆圈的行（或是圆圈本身）并选择 Enable Breakpoint 来完成。

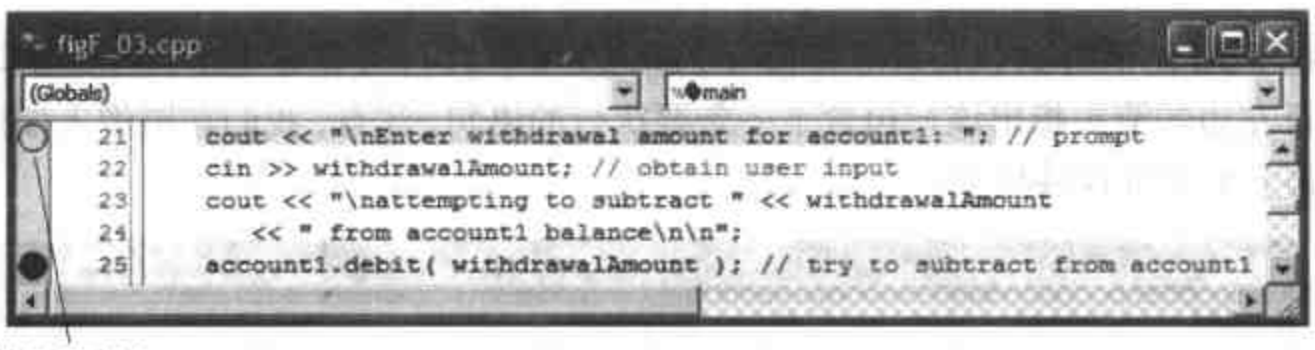


图 F.11　被禁用的断点

9. **删除断点。**要想删除不再需要的断点，可以右击设置了断点的代码行并选择 Remove Breakpoint，也可以通过点击空白指示条的圆圈删除断点。
10. **结束程序的执行。**选择 Debug➪Continue，使程序执行完毕。

在本节中，我们了解了如何启动调试器并设置断点以便在程序运行时检查代码的结果。也学习了程序在断点处暂停执行后如何继续执行以及如何禁用和删除断点。

F.3　Locals 窗口和 Watch 窗口

在前一节，我们了解到快速信息特性允许检查变量的值。在本节中，将学习使用局部变量窗口在程序运行时为变量赋新值。也将学习使用监视窗口检查更复杂的表达式的值。

1. **插入断点。**点击源代码第25行左边的空白指示条标注一个断点（见图F.12）。再点击源代码的第 28 行左边的空白指示条标注另一个断点。

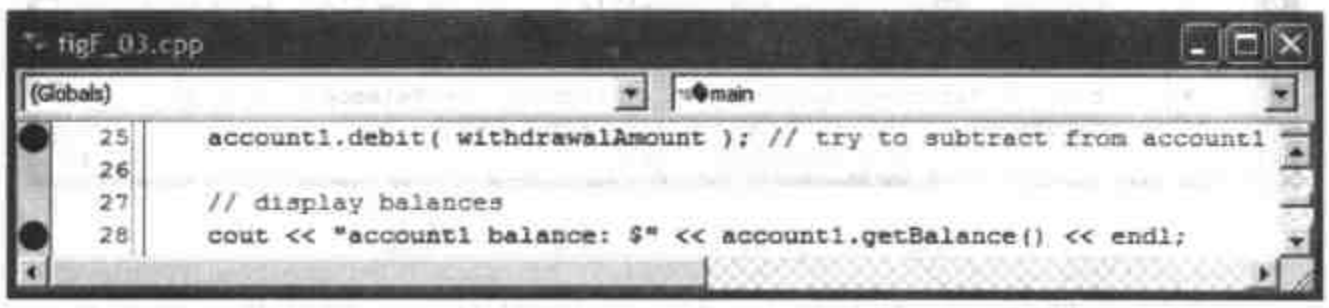


图 F.12　在第 25 行和第 28 行设置断点

2. **开始调试。**选择 Debug➪Start。在 Enter withdrawal amount for account1:提示下（见图 F.13）输入 13 并按回车键，以便程序读入刚才输入的值。程序执行直到遇到第 25 行处的断点。

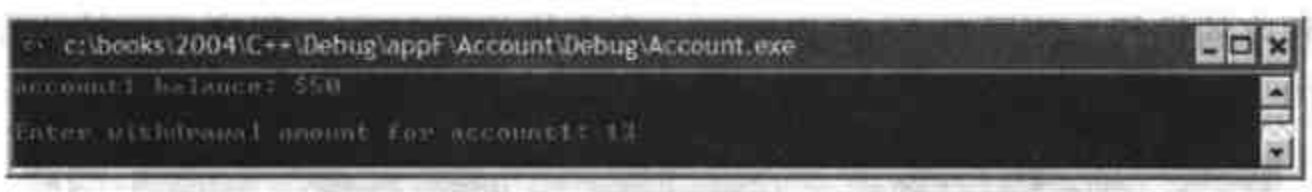


图 F.13　在遇到断点前输入提取数额

3. **暂停程序执行。**当程序运行到第25行时，Visual C++ .NET暂停程序执行并将程序转换到中断模式（见图 F.14）。此时，第 22 行的语句已将键入的值 13 输入到 withdrawalAmount，第23~24行中的语句已输出程序试图提取这笔金额，第25行的语句是下一条将要执行的语句。

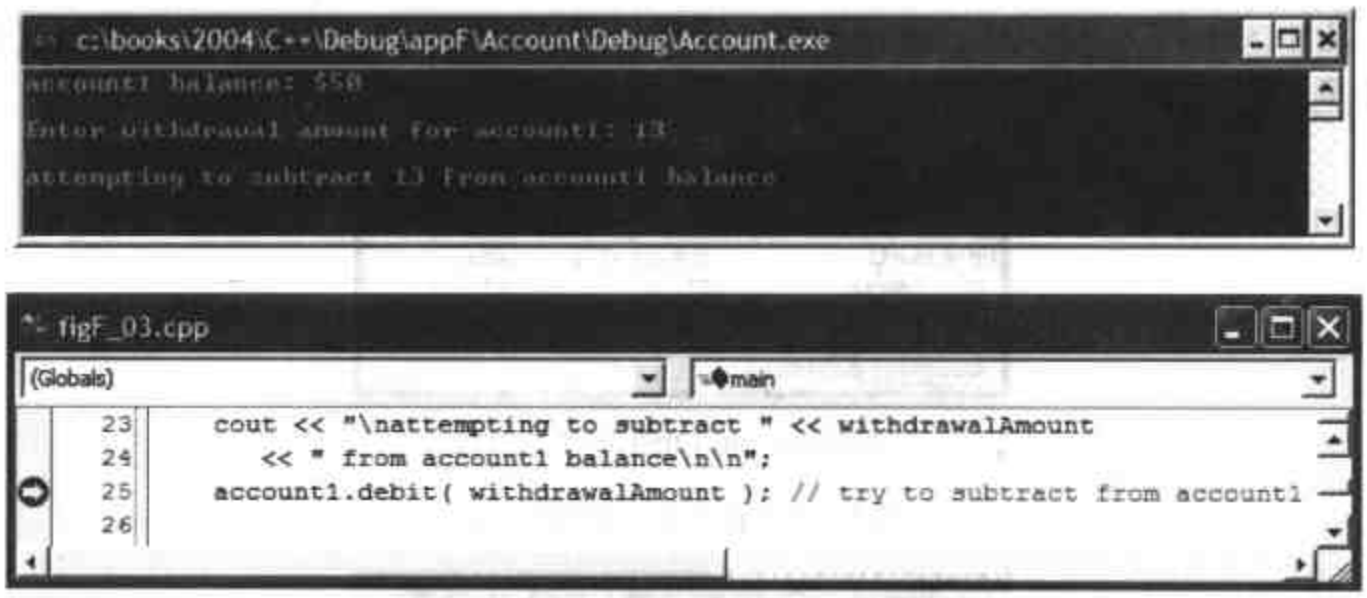


图 F.14　在第 25 行遇到断点时程序执行暂停

4. **检查数据**。一旦程序进入到中断模式，就可以使用调试器的 Locals 窗口探测局部变量的值。要想观察 Locals 窗口，选择 Debug➪Windows➪Locals。account1 和 withdrawalAmount（值为 13）的值会显示出来（见图 F.15）。

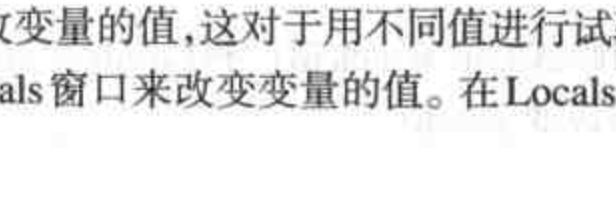


图 F.15　检查变量 withdrawalAmount

5. **求算术和布尔表达式的值**。Visual Studio .NET 允许使用 Watch 窗口计算算术和布尔表达式。共有四个不同的 Watch 窗口，但是我们只使用第一个。选择 Debug➪Windows➪Watch➪Watch1。在 Name 栏的第一行（初始应该是空的）键入(withdrawalAmount + 3)* 5，然后按回车键。注意，此 Watch 窗口可以求算术表达式的值。在本例中，它显示结果值 80（见图 F.16）。在 Watch 窗口 Name 栏的下一行，键入 withdrawalAmount == 3，然后按回车键。这个表达式判定 withdrawalAmount 里的值是否等于 3。包含符号 == 的表达式作为布尔表达式来处理。返回值为 false（见图 F.16），因为当前 withdrawalAmount 的值不是 3。

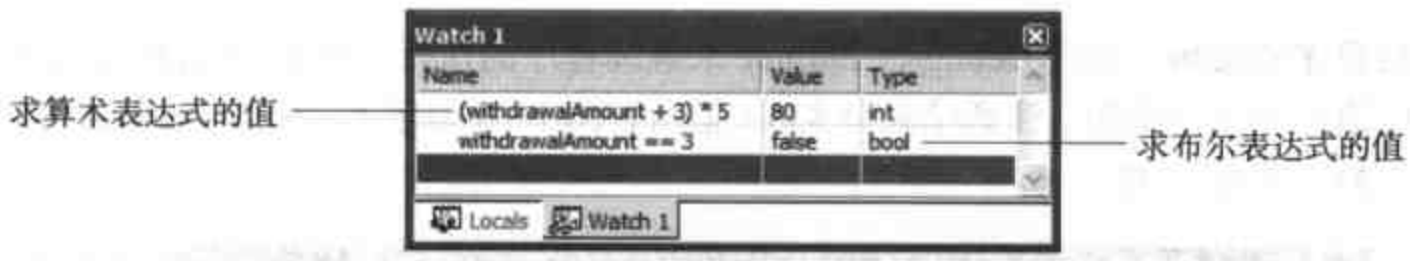
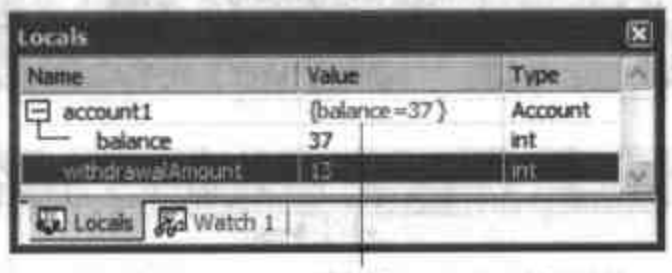


图 F.16　检查表达式的值

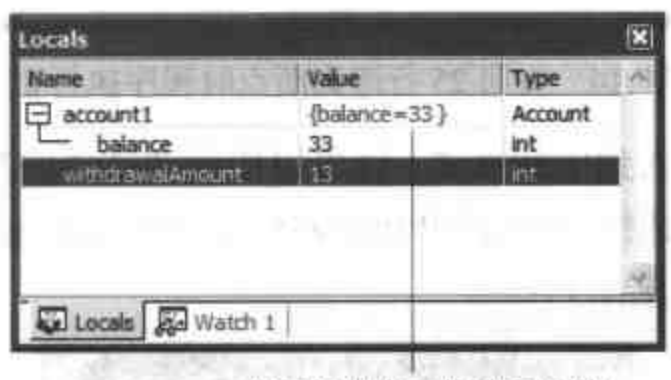
6. **恢复执行**。选择 Debug➪Continue 来恢复执行。执行第 25 行，从账户中减去所提的钱数，之后程序在第 28 行处再次暂停。选择 Debug➪Windows➪Locals。更新后的 account1 值现在以红色显示，表明它自上一次断点之后已被修改了（见图 F.17）。withdrawalAmount 中的值没有被标记为红色，因为它自上一次断点以来没有被修改过。点击 Locals 窗口 Name 栏 account1 左边的加号框。这允许你具体观察 account1 每个数据成员的值。
7. **修改值**。基于用户输入的值（是 13），程序输出的账户余额应为 37 美元。然而，可以使用调试器在程序执行中修改变量的值，这对于用不同值进行试验和定位程序中的逻辑错误是很有价值的。可以使用 Locals 窗口来改变变量的值。在 Locals 窗口中，点击 balance 行的 Value

域，选中37，键入33，然后按回车键。这样，调试器修改了balance的值并以红色显示新的值（见图F.18）。



变量account1的值显示为红色

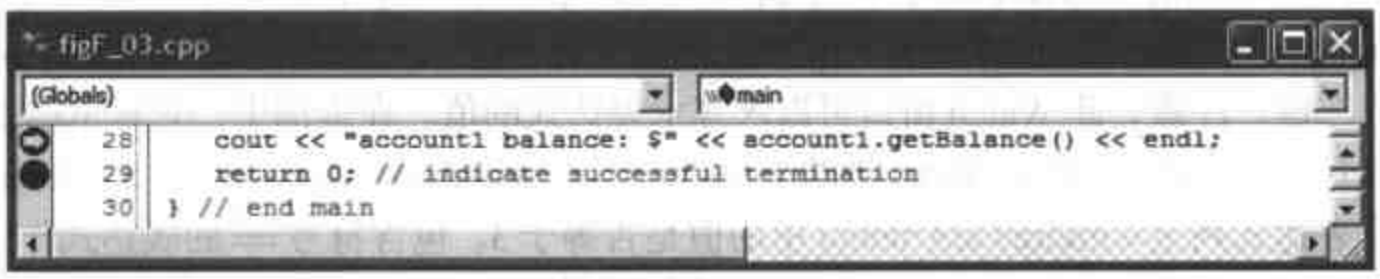
图F.17 显示局部变量的值



在调试器中已修改后的值

图F.18 修改一个变量的值

8. **在return语句中设置断点。**点击源代码第29行左边的空白指示条设置断点（见图F.19）。这可以防止程序显示结果之后立刻关闭。如果不设置这个断点，将不能在窗口关闭前看到程序的输出结果。



图F.19 在第29行设置断点

9. **观察程序的结果。**选择Debug⇨Continue来继续程序的执行。函数main执行直到第29行的返回语句并显示结果。注意结果是$33（见图F.20）。这表示上一步将balance值从所计算的值（37）改成了33。



图F.20 修改变量account1的值后显示的输出

10. **结束调试过程。**选择Debug⇨Stop Debugging，这将关闭Command Prompt窗口，删除所有剩余的断点。

在本节中，学习了如何使用调试器的Watch窗口和Locals窗口来求算术和布尔表达式的值，也学习了如何在程序执行过程中修改变量的值。

F.4　使用 Step Into、Step Over、Step Out 和 Continue 命令控制执行

有时候需要一行一行地执行程序来发现和确定逻辑错误。用这种方式执行部分程序可以帮助你核定函数代码执行是否正确。在本节中，将学习如何使用调试器来完成这项任务。本节学习的命令可以使你逐行地执行函数，或者一次性地执行函数的所有语句，或者只执行函数的剩余部分（如果之前已经运行了函数的某些语句）。

1. **设置断点。**点击空白指示条，在第 25 行处设置一个断点（见图 F.21）。

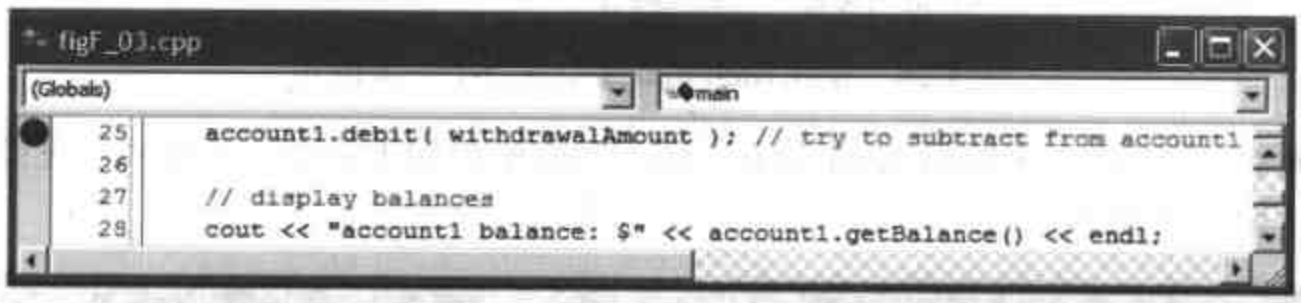


图 F.21　在程序中设置一个断点

2. **启动调试器。**选择 Debug⇨Start。在提示 Enter withdrawal amount for account1:时输入值 13。当程序运行到第 25 行时会暂停。
3. **使用 Step Into 命令。**Step Into 命令执行程序中的下一条语句（见下页图 F.24 中高亮的行）并立刻暂停。如果 Step Into 命令要执行的语句是一个函数调用，那么就转移控制到被调用的函数。Step Into命令允许你进入函数并通过单步地执行函数中的每条语句来确认函数的有效执行。选择 Debug⇨Step Into 进入 debit 函数（见图 F.22）。如果调试器不在第 33 行，再次选择 Debug⇨Step Into 来到达那行。

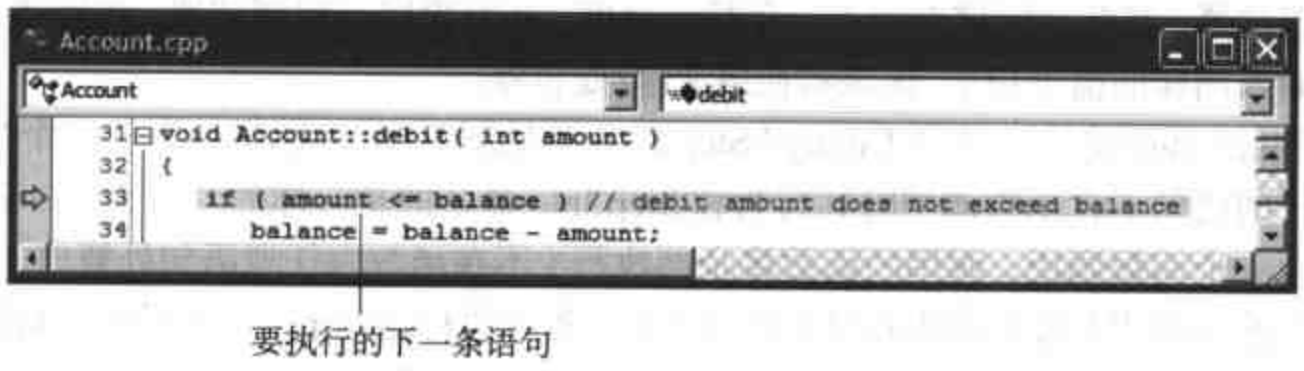


图 F.22　单步执行 debit 函数

4. **使用 Step Over 命令。**选择 Debug⇨Step Over 来执行当前的语句（见图 F.22 中的第 33 行）并控制转移到第 34 行（见图 F.23）。当下一条要执行的语句不含函数调用时，Step Over 命令如同 Step Into 命令。在第 10 步中将了解 Step Over 命令如何不同于 Step Into。

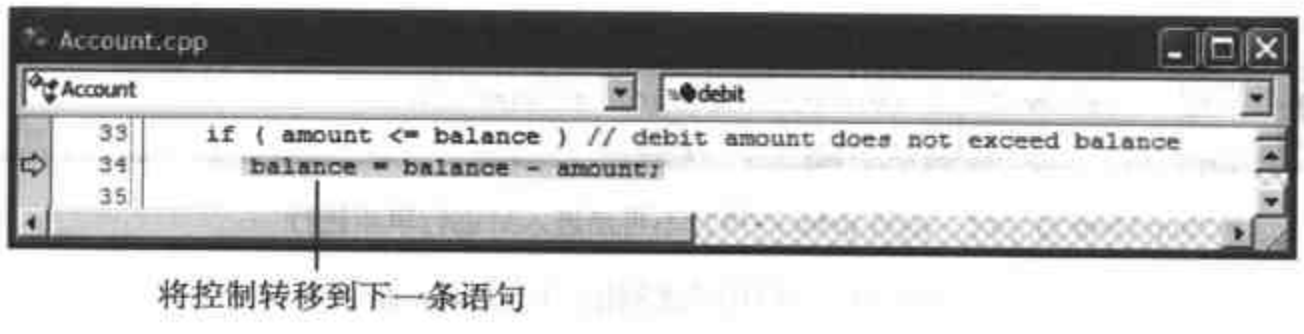


图 F.23　跨过 debit 函数中的一条语句

5. **使用Step Out命令。**选择Debug⇨Step Out来执行函数中剩下的语句并将控制返回给下一条要执行的语句（图F.3的第28行），这条语句包含了这个函数调用。通常，在较长的函数中，

你只想查看一些关键代码行，然后继续调试调用者的代码。当不想逐行地继续单步执行整个函数时，使用 Step Out 命令是很有用的。

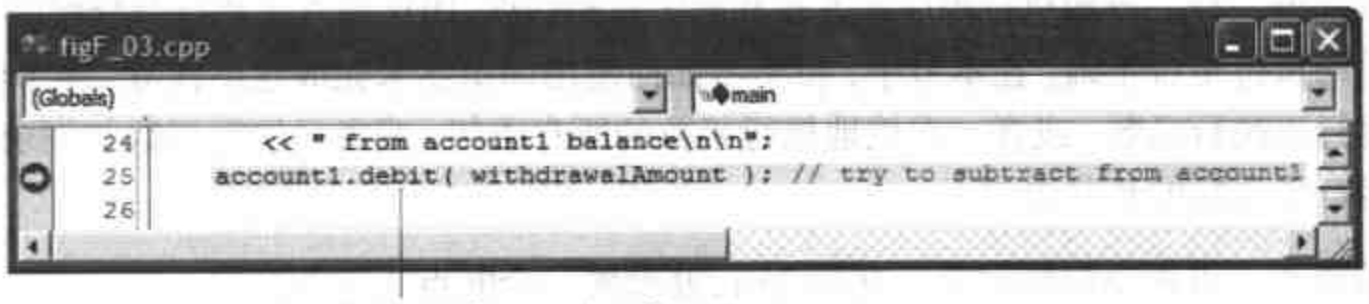


图 F.24 使用 Step Into 命令执行下一条语句

6. **设置断点。**在图 F.3 中的第 29 行 main 函数的 return 语句上设置一个断点（见图 F.25）。下一步会使用这个断点。

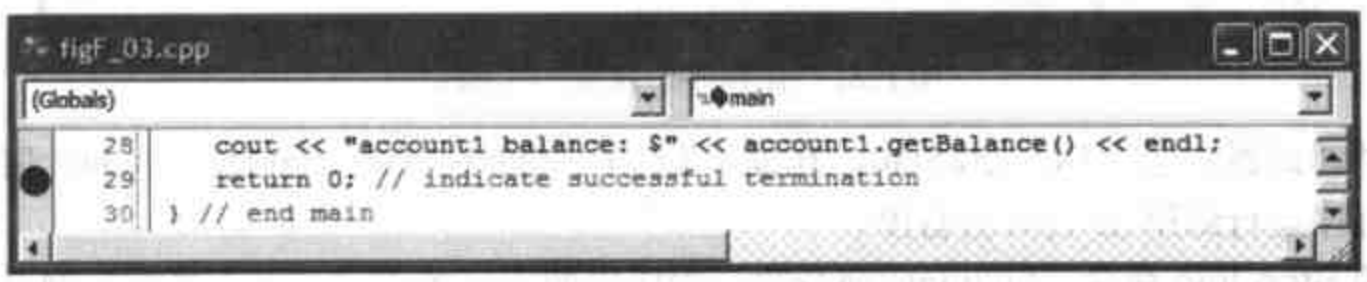


图 F.25 在程序中设置第二个断点

7. **使用 Continue 命令。**选择 Debug➪Continue 执行程序直到遇到第 29 行的断点。这个特性可以节省经逐行单步执行许多行代码才到下个断点所花费的时间。
8. **停止调试器。**选择 Debug➪Stop Debugging 来结束调试过程。这将关闭 Command Prompt 窗口。
9. **启动调试器。**在演示调试器的下一个特性之前，必须再次开启调试器。按照第 2 步启动调试器并输入同样的值（13）。调试器在第 25 行处暂停。
10. **使用 Step Over 命令。**选择 Debug➪Step Over（见图 F.26）。回忆一下，这个命令当下一条要执行的语句不含函数调用时其行为和 Step Into 命令的是一样的。如果下一条要执行的语句包含一个函数调用，调用的函数将全部执行（不在函数内任意语句处暂停），并且箭头前进到当前函数中（这条函数调用之后的）下一条可执行语句上。在本例中，调试器执行位于 main 函数中的第 25 行（见图 F.3）。第 25 行调用 debit 函数。调试器然后在第 28 行，即当前函数 main 的下一条可执行语句处暂停执行。

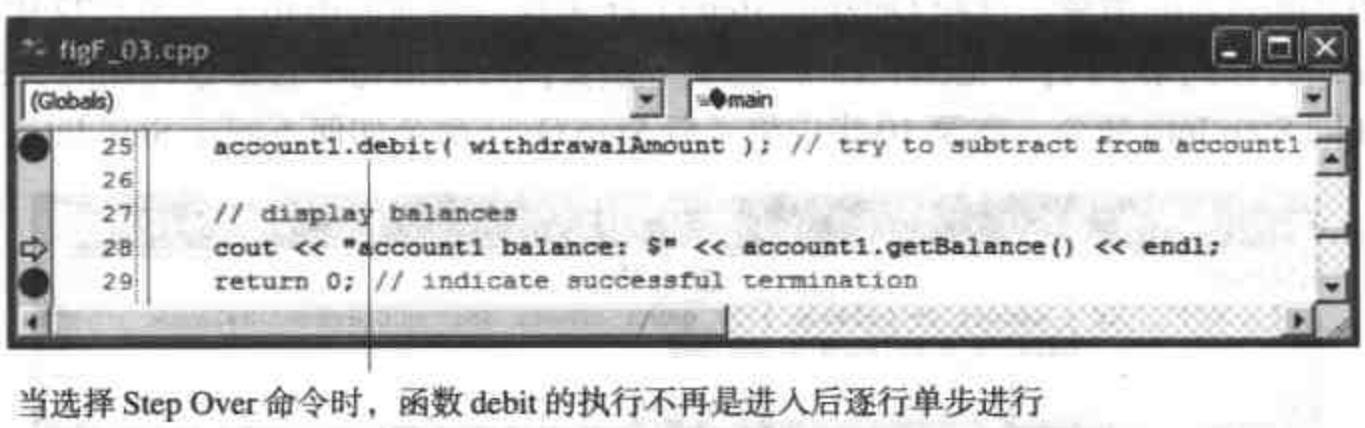


图 F.26 使用调试器的 Step Over 命令

11. **结束调试器。**选择 Debug➪Stop Debugging。这会关闭 Command Prompt 窗口并删除所有剩余断点。

在本节中，学习了如何使用调试器的 Step Into 命令调试在程序执行期间调用的函数。你已经看到了如何用 Step Over 命令跨过函数调用。你使用了 Step Out 函数来继续执行程序直到当前函数的末尾。也学习了使用 Continue 命令继续执行程序直到遇到另外一个断点或者程序退出。

F.5　Autos 窗口

在本节中，我们介绍 Autos（自动）窗口，该窗口显示前一条执行过的语句中使用的变量以及下一条要执行的语句中的变量。Autos 窗口允许你关注刚刚使用过的变量和下一条语句将要使用和修改的变量。

1. **设置断点。**点击第 14 行和第 22 行左边的空白指示条设置相应的断点（见图 F.27）。

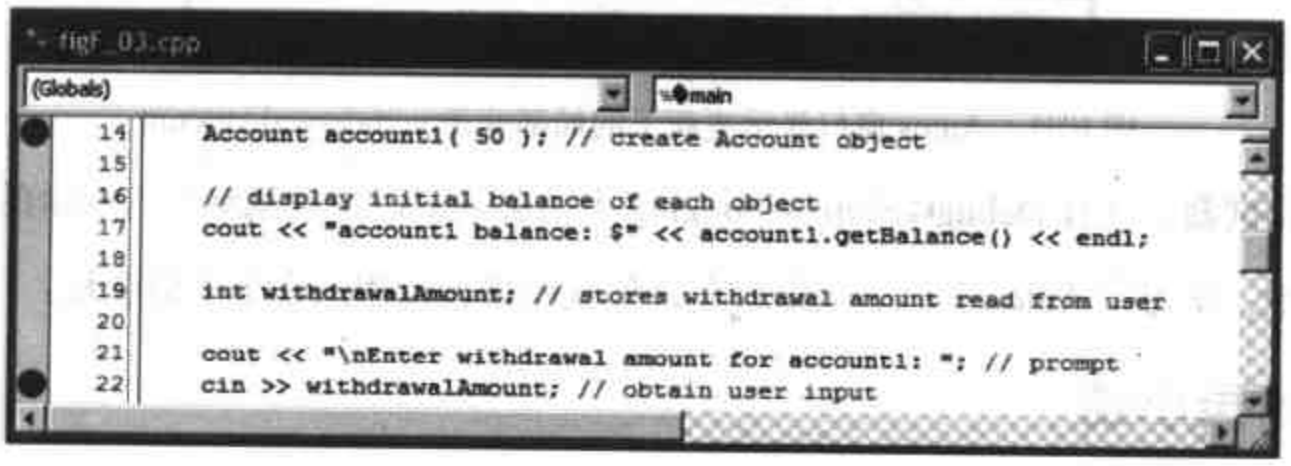


图 F.27　在程序中设置断点

2. **使用 Autos 窗口。**选择 Debug⇨Start 开始调试。当遇到第 14 行的断点时暂停执行，选择 Debug⇨Windows⇨Autos 打开 Autos 窗口（见图 F.28）。Autos 窗口允许你观察最近执行的语句使用的变量内容。这可以使你检查之前的语句执行是否正确。Autos 窗口也列出了下一条要执行的语句的值。注意，Autos 窗口列出了 account1 变量，它的值和类型。观察对象中的值可以使你证实程序计算这些变量是否正确。注意，account1 包含了一个大的负值。这个值在每次程序执行时都可能不同，它是account1 没有初始化时的值。这个不可预知（通常也是不受欢迎）的值也说明了为什么所有的 C++ 变量在使用前都要进行初始化的重要性。



图 F.28　Autos 窗口显示对象 account1 的状态

3. **使用 Step Over 命令。**选择 Debug⇨Step Over 来执行第 14 行。在 account1 值被初始化之后，Autos 窗口更新了这个值（见图 F.29）。account1 的值用红色显示，表示它已被修改。



图 F.29　Autos 窗口显示初始化后对象 account1 的状态

4. **继续执行。**选择 Debug⇨Continue。程序执行将暂停在第 22 行的断点处。Autos 窗口（见图 F.30）显示还没初始化的局部变量 withdrawalAmount 被设置了，它有一个很大的负值。



图 F.30 Autos 窗口显示局部变量 withdrawalAmount

5. **输入数据。**选择Debug➪Step Over执行第22行。当程序提示输入时，输入提取的金额。Autos窗口（见图F.31）将用输入的值更新局部变量 withdrawalAmount 的值。注意，Autos 窗口的第一行包含用于输入数据的 istream 对象（cin）。

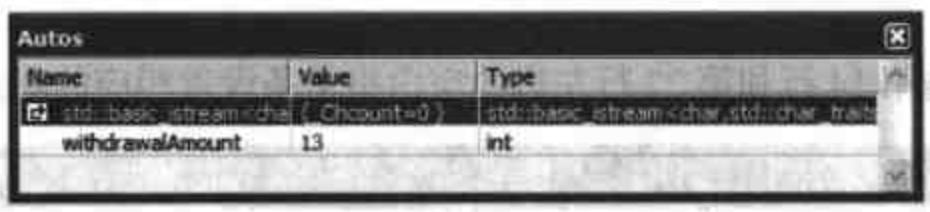


图 F.31 Autos 窗口显示更新了的局部变量 withdrawalAmount

6. **结束调试器。**选择 Debug➪Stop Debugging 来结束调试过程并删除所有剩余的断点。

在本节中，学习了 Autos 窗口。该窗口允许你观察最近使用的命令中的变量。

F.6 本附录小结

在本附录中，学习了在 Visual Studio.NET 调试器中如何插入、禁用和删除断点。断点允许你暂停程序的执行，因此可以检查变量的值。这个功能将有助于你发现和确定程序中的逻辑错误。我们明白了如何使用Locals窗口和Watch窗口来检查表达式的值及如何修改变量的值。还学习了如何使用调试器命令 Step Into、Step Over、Step Out 和 Continue，以判断函数是否正确执行了。最后，学习了如何使用 Autos 窗口检查最近使用的语句和下一条命令中的变量。

摘要

- 绝大多数的 C++ 编译器厂商都提供称为调试器的软件，使你可以监控程序的执行，以便定位和消除逻辑错误。
- 断点是可以在任何可执行代码行设置的记号。当程序执行遇到断点时，执行就会暂停。
- 调试器默认是激活的。如果它没有被激活，就不得不修改解决方法配置组合框的设定。
- 要想插入断点，可以点击代码行左边的空白指示条，或者右击那条代码行并选择 Insert Breakpoint。当在点击处出现实心栗色圆圈时，表示断点已被设置。
- 当程序运行时，在任何包含断点的代码处程序执行都会暂停。这称为进入中断模式，IDE的标题栏将显示[break]。
- 黄色箭头表明该行包含下一条要执行的语句。
- 当将鼠标指针放在变量名上时，变量存储的值将显示在快速信息框中。
- 要想禁用断点，右击设置了断点的代码行并选择Disable Breakpoint。被禁用的断点显示为空心栗色圆圈。
- 要想删除不再需要的断点，可以右击设置了断点的代码行并选择 Remove Breakpoint，也可以点击空白指示条中的栗色圆圈来删除断点。
- 一旦程序进入中断模式，可以使用调试器的 Locals 窗口来查看变量的值。要观察 Locals 窗口，选择 Debug➪Windows➪Locals。

附录G　使用GNU C++调试器



那么我将捕风捉影了。

——William Shakespeare

我们生来就易犯错误，遗传密码决定如此。

——Lewis Thomas

我们能够预料之事很少出现，而我们不期望的事却总是发生（计划赶不上变化）。

——Benjamin Disraeli

他可以跑，但是不可以躲起来。

——Joe Louis

一件事可以说明他是错误的，而另一件事又可以证明他是正确的。

——John Locke

学习目标

在本附录中将学习：

- 在调试器中用run命令来运行程序
- 使用break命令设置断点
- 使用continue继续执行程序
- 使用print命令求表达式的值
- 使用set命令在程序的执行过程中改变变量的值
- 使用step、finish和next命令控制程序的执行
- 使用watch命令观察数据成员在程序的执行过程中是如何被修改的
- 使用delete删除断点或监视点

提纲

G.1　简介　　G.2　断点及run、stop、continue和print命令
G.3　print和set命令　　G.4　使用step、finish和next命令控制程序的执行
G.5　watch命令　　G.6　本附录小结

摘要 | 术语 | 自测题 | 自测题答案

G.1　简介

在第2章，我们知道了错误的类型有两种——编译错误和逻辑错误，并且学习了如何从代码中消除编译错误。逻辑错误也称为程序缺陷，并不会使程序编译失败，但是在程序运行时，它的确使

程序产生错误的结果。GNU包含了一种叫调试器（debugger）的软件，它能够让你监控程序的整个执行过程，因此可以定位和消除逻辑错误。

这种调试器将成为你最为重要的程序开发工具之一。很多IDE（集成开发环境）提供了自己的调试器（类似于GNU的调试器）或者为GNU调试器提供了图形用户界面。本附录展示了GNU调试器的主要特征。附录F讨论了Visual Studio .NET调试器的特征和性能。我们提供了一些免费的Dive Into系列出版物来帮助学生和教师，使他们能够熟悉各种开发工具中所提供的调试器。这些出版物能够在随书附赠的光盘中得到，也可以从www.deitel.com/books/downloads下载。

G.2　断点与 run、stop、continue 和 print 命令

我们就从研究断点（breakpoints）开始进行调试器的学习。所谓断点是指可以在任何可执行代码行上设置的标记。当程序执行遇到断点时，执行便会暂停，允许你通过检查变量的值来确定是否存在逻辑错误。举例来说，在计算过程中，有一个变量用来存储结果，那么你可以通过检查这个变量的值来判断计算是否正确。注意，如果试着在某函数的一条不可执行的代码行（比如注释）上设置断点，你会发现该断点实际上是被设置在了函数中下一条可执行的代码上。

为了举例说明调试器的特性，我们使用图G.3中的程序，该程序创建并操作类Account（见图G.1和图G.2）的一个对象，程序从main函数（见图G.3中的第12~30行）开始执行。第14行创建了一个Account的对象，同时把它的数据成员balance初始化为50.00美元。Account的构造函数（见图G.3中的第10~22行）接受一个参数，指定了balance的初始值。图G.3的第17行使用Account的成员函数getBalance，输出余额的初始值。第19行声明了一个名为withdrawalAmount的局部变量，它存储从用户那里读入的提取金额数。第21行提示用户输入提取金额，第22行将此金额输入到变量withdrawalAmount。第25行使用成员函数debit从这个Account对象的balance中减去提取金额的值。最后，第28行显示新的balance。

```
1  // Fig. G.1: Account.h
2  // Definition of Account class.
3
4  class Account
5  {
6  public:
7     Account( int ); // constructor initializes balance
8     void credit( int ); // add an amount to the account balance
9     void debit( int ); // subtract an amount from the account balance
10    int getBalance(); // return the account balance
11 private:
12    int balance; // data member that stores the balance
13 }; // end class Account
```

图 G.1　类 Account 的头文件

```
1  // Fig. G.2: Account.cpp
2  // Member-function definitions for class Account.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Account.h" // include definition of class Account
8
9  // Account constructor initializes data member balance
10 Account::Account( int initialBalance )
11 {
12    balance = 0; // assume that the balance begins at 0
```

图 G.2　类 Account 的定义

```
13
14    // if initialBalance is greater than 0, set this value as the
15    // balance of the Account; otherwise, balance remains 0
16    if ( initialBalance > 0 )
17       balance = initialBalance;
18
19    // if initialBalance is negative, print error message
20    if ( initialBalance < 0 )
21       cout << "Error: Initial balance cannot be negative.\n" << endl;
22 } // end Account constructor
23
24 // credit (add) an amount to the account balance
25 void Account::credit( int amount )
26 {
27    balance = balance + amount; // add amount to balance
28 } // end function credit
29
30 // debit (subtract) an amount from the account balance
31 void Account::debit( int amount )
32 {
33    if ( amount <= balance ) // debit amount does not exceed balance
34       balance = balance - amount;
35
36    else // debit amount exceeds balance
37       cout << "Debit amount exceeded account balance.\n" << endl;
38 } // end function debit
39
40 // return the account balance
41 int Account::getBalance()
42 {
43    return balance; // gives the value of balance to the calling function
44 } // end function getBalance
```

图 G.2 类 Account 的定义（续）

```
1  // Fig. G.3: figG_03.cpp
2  // Create and manipulate Account objects.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  // include definition of class Account from Account.h
9  #include "Account.h"
10
11 // function main begins program execution
12 int main()
13 {
14    Account account1( 50 ); // create Account object
15
16    // display initial balance of each object
17    cout << "account1 balance: $" << account1.getBalance() << endl;
18
19    int withdrawalAmount; // stores withdrawal amount read from user
20
21    cout << "\nEnter withdrawal amount for account1: "; // prompt
22    cin >> withdrawalAmount; // obtain user input
23    cout << "\nattempting to subtract " << withdrawalAmount
24       << " from account1 balance\n\n";
25    account1.debit( withdrawalAmount ); // try to subtract from account1
26
27    // display balances
28    cout << "account1 balance: $" << account1.getBalance() << endl;
29    return 0; // indicate successful termination
30 } // end main
```

图 G.3 为了调试而测试类

在下面的步骤中，将使用断点和各种调试命令来检查在图G.3中第19行声明的变量withdrawalAmount的值。

1. **编译程序以便调试。**GNU调试器只会对编译时附加了-g参数而产生的可执行文件起作用，因为-g编译选项会产生一些信息，调试器利用它们来帮助你调试程序。用-g命令行选项来编译图G.3中的程序，键入：g++ -g -o figG_03 figG_03.cpp Account.cpp。
2. **启动调试器。**键入gdb figG_03（见图G.4），这条命令将启动GNU调试器并显示可以输入命令的（gdb）提示符。

```
~/Debug$ gdb figG_03
GNU gdb 6.1-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...Using host libthread_db library "/
lib/libthread_db.so.1".

(gdb)
```

图G.4　启动调试器以运行程序

3. **在调试器中运行程序。**键入run命令（见图G.5），以便在调试器中运行程序。如果在调试器中运行程序之前没有设置任何断点，那么程序将顺利地完成运行。

```
(gdb) run
Starting program: /home/student/Debug/figG_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37

Program exited normally.
(gdb)
```

图G.5　在没有设置断点的情况下运行程序

4. **使用GNU调试器插入断点。**我们在程序的特定代码行插入一个断点。在这些步骤中所使用到的行号来源于图G.3中的源代码。通过键入break 17在源代码的第17行设置一个断点。break命令在命令后行号指定的行上插入断点。可以根据需要设置尽可能多的断点。每个断点依据它们创建的次序标识出来。第一个被创建的断点叫做Breakpoint 1。通过键入break 25在第25行设置另一个断点（见图G.6）。当程序运行时，它将在任何设置了断点的行上暂停执行，并且使程序进入中断模式（break mode）。甚至可以在调试过程开始后设置断点。注意，如果不知道代码的行号，可以使用list命令来输出带行号的代码。读者在gdb提示符下用help命令可以获得更多关于list命令的信息。
5. **运行程序并开始调试过程。**键入run命令执行程序并开始调试过程（见图G.7）。当程序运行到第17行的断点时暂停执行。此时，调试器会提醒你程序已经执行到断点处，同时显示第17行上的代码。此代码行包含了下一条要执行的语句。

```
(gdb) break 17
Breakpoint 1 at 0x80487d8: file figG_03.cpp, line 17.
(gdb) break 25
Breakpoint 2 at 0x8048871: file figG_03.cpp, line 25.
(gdb)
```

图 G.6 在程序中设置两个断点

```
(gdb) run
Starting program: /home/student/Debug/figG_03

Breakpoint 1, main () at figG_03.cpp:17
17          cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

图 G.7 运行程序一直到第一个断点处

6. **使用continue命令恢复程序的执行。**键入continue。continue命令使处于暂停状态的程序继续运行直到遇到下一个断点（第25行）。在提示符下输入13。当程序执行到第二个断点处时，调试器会提醒你（见图G.8）。注意程序figG_03的正常输出夹杂在调试器发出的信息中间。

```
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance


Breakpoint 2, main () at figG_03.cpp:25
25          account1.debit( withdrawalAmount ); // try to subtract from
account1
(gdb)
```

图 G.8 继续执行程序，直到第二个断点处

7. **检查变量的值。**键入print withdrawalAmount，以显示当前存储在变量withdrawalAmount中的值（见图G.9）。print命令允许你窥探程序中变量的值，有助于查找并消除代码中的逻辑错误。注意显示的结果为13，这正是图G.3的第22行中读入并分配给变量withdrawalAmount的值。使用print命令输出对象account1的内容。当一个对象通过调试器的print命令输出时，输出的是用花括号（{}）括起来的该对象的数据成员。在这个例子中，只有一个数据成员，即balance，它的值是50。

```
(gdb) print withdrawalAmount
$1 = 13
(gdb) print account1
$2 = {balance = 50}
(gdb)
```

图 G.9 显示变量的值

8. **使用快捷变量。**当使用print命令时，结果存储于快捷变量，例如$1中。快捷变量属于临时变量，命名方式是一个$后面接一整型数字。它在调试程序的过程中显示变量值时由调试器创建。快捷变量在调试过程中能够被用来求算术与布尔表达式的值。键入print $1，调试器将显示$1的值（见图G.10），它包含了变量withdrawalAmount的值。注意，显示$1的值将会创建一个新的快捷变量——$3。

```
(gdb) print $1
$3 = 13
(gdb)
```

图G.10　显示一个快捷变量

9. **继续程序的执行。**键入continue使程序继续执行。因为调试器不再遇到其他断点了，所以程序一直执行至结束（见图G.11）。

```
(gdb) continue
Continuing.
account1 balance: $37

Program exited normally.
(gdb)
```

图G.11　结束程序的执行

10. **删除断点。**通过键入info break命令可以列出程序中的所有断点。如果要删除一个断点，可以键入delete后接一个空格符，然后是要删除的断点的编号。键入delete 1可以删除第一个断点。用同样方法删除第二个断点。现在，如果再用info break命令来列出程序中剩余的断点，那么调试器应指出没有设置的断点了（见图G.12）。

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080487d8 in main at figG_03.cpp:17
        breakpoint already hit 1 time
2   breakpoint     keep y   0x08048871 in main at figG_03.cpp:25
        breakpoint already hit 1 time
(gdb) delete 1
(gdb) delete 2
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

图G.12　观察并删除断点

11. **在没有断点的情况下执行程序。**键入run执行程序。在提示符下输入13。因为你已经成功地删除了两个断点，所以程序并没有进入中断模式，而是正常显示输出（见图G.13）。
12. **使用quit命令。**使用quit命令结束调试过程（见图G.14）。这条命令使调试器结束调试。

在这一节中，学习了如何用gdb命令来启动调试器，以及如何用run命令来运行程序。你看到了怎样在main函数中的某特定行上设置一个断点。break命令能够用来在另一个文件中的某行或特定的函数上设置断点。键入break，然后依次是文件名、一个冒号和行号，就可以在另一个文件的

某一行设置一个断点。键入break，然后是一个函数名，这样，每当这个函数被调用时，调试器都会进入中断模式。

```
(gdb) run
Starting program: /home/student/Debug/figG_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37

Program exited normally.
(gdb)
```

图 G.13　程序在没有断点被设置的情况下执行

```
(gdb) quit
~/Debug$
```

图 G.14　使用 quit 命令退出调试器

在这一节里，还可以知道 help list 命令将提供更多关于 list 命令的信息。如果对调试器或者任何命令有任何疑问的话，可以键入 help 或者 help 后接命令名来获得更多的帮助信息。

最后，学习了可以用 print 命令来检查变量，用 delete 命令来删除断点。也知道了如何使用 continue 命令使程序遇到一个断点后继续执行，以及怎样用 quit 命令来结束调试器。

G.3　print 和 set 命令

在前一节中，学习了怎样在程序的执行期间使用调试器的 print 命令来检查变量的值。在这一节中，将学习怎样使用 print 命令来检查更复杂表达式的值。也将学习 set 命令，它允许程序员给变量赋予新的值。我们假设你正在研究学习包含本附录实例的目录，并且已经用 -g 编译器选项编译了这些实例，以便于调试。

1. **开始调试。**键入 gdb figG_03 启动 GNU 调试器。
2. **插入断点。**键入 break 25，在源代码的第 25 行设置一个断点（见图 G.15）。

```
(gdb) break 25
Breakpoint 1 at 0x8048871: file figG_03.cpp, line 25.
(gdb)
```

图 G.15　在程序中设置一个断点

3. **运行程序直至遇到一个断点。**键入 run 启动调试过程（见图 G.16），这使得 main 函数开始执行，直到遇到第 25 行的断点。此时程序暂停执行并被切换到中断模式。第 25 行中的语句是下一条将要被执行的代码。
4. **求算术与布尔表达式的值。**回顾 G.2 节，一旦程序进入中断模式，就可以使用调试器的 print 命令来查看程序中变量的值。另外，还可以使用 print 命令来计算算术与布尔表达式的值。键入 print withdrawalAmount - 2。注意，print 命令返回值 11（见图 G.17），但是，实际上这条

命令并没有改变变量 withdrawalAmount 的值。键入 print withdrawalAmount == 11，含有符号==的表达式按照布尔表达式来处理。因为变量 withdrawalAmount 的当前值并不是 11（它的值仍为 13），所以返回值为 false（见图 G.17）。

```
(gdb) run
Starting program: /home/student/Debug/figG_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance


Breakpoint 1, main () at figG_03.cpp:25
25          account1.debit( withdrawalAmount ); // try to subtract from
account1
(gdb)
```

图 G.16　运行程序直至遇到在第 25 行的断点

```
(gdb) print withdrawalAmount - 2
$1 = 11
(gdb) print withdrawalAmount == 11
$2 = false
(gdb)
```

图 G.17　通过调试器显示表达式

5. **修改变量的值。**调试器允许在程序的执行过程中修改变量的值。这对于用不同的值进行试验和定位程序中的逻辑错误是非常有用的。可以使用调试器的 set 命令改变变量的值。键入 set withdrawalAmount = 42，调试器就会改变变量 withdrawalAmount 的值。键入 print withdrawalAmount 显示它的新值（见图 G.18）。

```
(gdb) set withdrawalAmount = 42
(gdb) print withdrawalAmount
$3 = 42
(gdb)
```

图 G.18　在中断模式下设置变量的值

6. **观察程序的结果。**键入continue继续执行程序。图 G.3 中的第 25 行的代码将变量 withdrawal-Amount 的值传递给 Account 的成员函数 debit，然后 main 函数将显示新的余额值。注意，结果是 $8（见图 G.19）。这表明上一步将变量 withdrawalAmount 的值从 13（初始值）改为了 42。

```
(gdb) continue
Continuing.
account1 balance: $8

Program exited normally.
(gdb)
```

图 G.19　在程序执行中使用一个修改过的变量

7. **使用 quit 命令。**使用 quit 命令结束调试过程（见图 G.20），此命令导致调试器终止。

```
(gdb) quit
~/Debug$
```

图 G.20 使用 quit 命令退出调试器

在这一节中，学习了如何使用调试器的 print 命令来求算术与布尔表达式的值，也学习了如何使用 set 命令在程序的运行过程中修改变量的值。

G.4 使用 step、finish 和 next 命令控制程序执行

有时候，需要一行一行地执行程序来发现和定位错误。通过这种方式执行部分程序能帮助你判定函数中的代码执行是否正确。在这一节中，将学习如何使用调试器来完成这项任务。在本节中学到的命令能让你逐行地执行一个函数，或者一次性地执行函数中的所有语句，或者仅仅执行函数中的剩余语句（如果之前已经执行了函数中的某些语句）。我们再次假设你正在研究包含本附录实例的目录，并且已经用 -g 编译器选项编译了这些实例，以便于调试。

1. **开始调试。**键入 gdb figG_03 启动 GNU 调试器。
2. **设置断点。**键入 break 25，在源代码的第 25 行设置一个断点。
3. **运行程序。**键入 run 运行程序。在程序输出两条消息后，调试器提醒已经到达一个断点并且显示第 25 行的代码。之后，调试器和程序暂停并且等待输入下一条命令。
4. **使用 step 命令。**step 命令用来执行程序中的下一条语句。如果下一条要执行的语句是一个函数调用，则控制权转移到被调用函数。step 命令使你能够进入到一个函数，对这个函数的语句逐条进行分析。例如，可以用 print 和 set 命令来观察并修改函数内部的变量。键入 step，进入类 Account 的成员函数 debit（见图 G.2）。调试器将提示此步已经完成并显示下一条可执行语句（见图 G.21）——在这个例子中，是类 Account 的第 33 行（见图 G.2）。

```
(gdb) step
Account::debit (this=0xbffffd70, amount=13) at Account.cpp:33
33          if ( amount <= balance ) // debit amount does not exceed balance
(gdb)
```

图 G.21 使用 step 命令进入一个函数

5. **使用 finish 命令。**在进入成员函数 debit 后，键入 finish。这条命令将执行函数中的剩余语句并将控制权返回到函数的调用处。finish 命令在执行了成员函数 debit 中的剩余语句后，在 main 函数的第 28 行暂停（见图 G.22）。在一些比较长的函数中，可能需要查看代码中的一些关键行，然后继续对调用者的代码进行调试。当不希望继续逐条语句地对整个函数进行单步调试时，finish 命令是非常有用的。

```
(gdb) finish
Run till exit from #0  Account::debit (this=0xbffffd70, amount=13)
    at Account.cpp:33
main () at figG_03.cpp:28
28          cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

图 G.22 使用 finish 命令完成一个函数的执行并返回调用函数

6. **使用continue命令继续执行程序。**键入continue命令继续执行程序。因为没有再遇到其他断点，所以程序执行直至结束。
7. **再次运行程序。**在调试过程中设置断点后，所有的断点会一直存在并被管理——即使在程序执行完以后。在第2步中设置的断点在下一次程序执行时仍然存在。键入run执行程序。如同第3步一样，程序一直运行直至遇到第25行的断点，然后调试器暂停并等待下一条命令（见图G.23）。

```
(gdb) run
Starting program: /home/student/Debug/figG_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance


Breakpoint 1, main () at figG_03.cpp:25
25          account1.debit( withdrawalAmount ); // try to subtract from
account1
(gdb)
```

图G.23　重新运行程序

8. **使用next命令。**键入next。这个命令和step相似，只是当下一条要执行的语句中含有函数调用时有所不同——这时，程序会执行整个被调用函数，然后停留在该函数调用之后的下一条可执行语句行上（见图G.24）。在第4步中，step命令使执行进入被调用函数。在这个例子中，next命令使执行Account的成员函数debit执行，然后调试器暂停在第28行。

```
(gdb) next
28          cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

图G.24　使用next命令执行完整个函数

9. **使用quit命令。**使用quit命令结束调试过程（见图G.25）。如果程序此刻正在运行，该命令将导致运行中的程序立即结束，而并不是继续执行main函数中的剩余语句。

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
~/Debug$
```

图G.25　使用quit命令退出调试器

在这一节中，学习了如何使用调试器的step和finish命令来调试程序执行期间被调用的函数，看到了如何使用next命令来跨过一个函数调用，还学习了如何使用quit命令来结束调试过程。

G.5　watch命令

在这一节中，我们将介绍监视命令（watch command），它能让调试器监视数据成员。当数据成员准备改变时，调试器会发出提示。在本节中，将学习如何使用watch命令，在程序的执行过程中观察Account对象的数据成员balance的改变情况。

1. **启动调试器。**键入 gdb figG_03，启动 GNU 调试器。
2. **运行程序。**键入 break 14，在第 14 行设置一个断点。用 run 命令开始运行程序。调试器和程序将会在第 14 行的断点处暂停（见图 G.26）。

```
(gdb) run
Starting program: /home/student/Debug/figG_03

Breakpoint 1, main () at figG_03.cpp:14
14          Account account1( 50 ); // create Account object
(gdb)
```

图 G.26　运行程序直至遇到第一个断点

3. **监视类的数据成员。**键入 watch account1.balance，在 account1 的数据成员 balance 上设置一个监视点（见图 G.27）。这个监视点标记为 watchpoint 2，因为监视点与断点在一起编号。在调试器执行期间，可以对当前作用域中对象的任何变量或数据成员设置监视点。当被监视变量的值改变时，调试器会进入中断模式并提醒你变量值已经改变了。

```
(gdb) watch account1.balance
Hardware watchpoint 2: account1.balance
(gdb)
```

图 G.27　在一个数据成员上设置监视点

4. **继续执行程序。**用 step 命令单步执行类 Account 的构造函数。调试器将显示图 G.2 中的第 12 行——它是该构造函数中的第一行。再次用 step 命令执行这行代码。这时，调试器会提示数据成员 balance 的值将要改变（见图 G.28）。当程序开始时，创建了类 Account 的一个实例。当此对象的构造函数执行时，数据成员 balance 首先被赋值为 0。此时调试器提醒你 balance 的值已经改变了。

```
(gdb) step
Account (this=0xbffffd70, initialBalance=50) at Account.cpp:12
12          balance = 0; // assume that the balance begins at 0
(gdb) step
Hardware watchpoint 2: account1.balance

Old value = 1073833120
New value = 0
Account (this=0xbffffd70, initialBalance=50) at Account.cpp:16
16          if ( initialBalance > 0 )
(gdb)
```

图 G.28　单步调试构造函数

5. **结束构造函数。**键入 step 执行第 16 行代码，然后再次键入 step 以执行第 17 行代码。调试器将提醒你数据成员 balance 的值从 0 变成了 50（见图 G.29）。
6. **从账户account中提取金额。**键入continue继续执行程序并在提示符下输入要提取金额的值。程序正常执行。图 G.3 中的第 25 行调用 Account 的成员函数 debit，让它从 Account 对象的数据成员 balance 中减去指定的 amount 值。debit 函数内部，即图 G.2 的第 34 行改变 balance 的值。调试器通知你这个改变并进入中断模式（见图 G.30）。

```
(gdb) step
17              balance = initialBalance;
(gdb) step
Hardware watchpoint 2: account1.balance

Old value = 0
New value = 50
Account (this=0xbffffd70, initialBalance=50) at Account.cpp:20
20          if ( initialBalance < 0 )
(gdb)
```

图G.29　到达一个监视点提示

```
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

Hardware watchpoint 2: account1.balance

Old value = 50
New value = 37
0x08048a01 in Account::debit (this=0xbffffd70, amount=13) at Account.cpp:34
34              balance = balance - amount;
(gdb)
```

图G.30　当变量值改变时进入中断模式

7. **继续执行程序。**键入continue，因为程序不再试图改变balance的值，因此main函数一直执行至结束。调试器删除在对象account1的数据成员balance上的监视点，因为当main函数结束时，该变量已经走出了它的作用域。删除监视点会导致调试器进入中断模式。再次键入continue以完成程序的执行（见图G.31）。

```
(gdb) continue
Continuing.
end of function
account1 balance: $37

Watchpoint 2 deleted because the program has left the block in
which its expression is valid.
0x4012fa65 in exit () from /lib/libc.so.6
(gdb) continue
Continuing.

Program exited normally.
(gdb)
```

图G.31　继续执行直至程序结尾

8. **重启调试器并在变量上重设监视点。**键入run以重新启动调试器。键入watch account1.balance在对象account1的数据成员上再次设置一个监视点。这个监视点被标记为watchpoint 3。键入continue继续程序的执行（见图G.32）。

```
(gdb) run
Starting program: /home/student/Debug/figG_03

Breakpoint 1, main () at figG_03.cpp:14
14          Account account1( 50 ); // create Account object
(gdb) watch account1.balance
Hardware watchpoint 3: account1.balance
(gdb) continue
Continuing.
Hardware watchpoint 3: account1.balance

Old value = 1073833120
New value = 0
Account (this=0xbffffd70, initialBalance=50) at Account.cpp:16
16          if ( initialBalance > 0 )
(gdb)
```

图 G.32　在数据成员变量上重新设置一个监视点

9. **从数据成员上删除监视点。**假设你只希望在部分程序的执行中监视一个数据成员。可以键入 delete 3 删除调试器设在变量 balance 上的监视点（见图 G.33）。键入 continue，程序将执行到结束而不会再进入中断模式。

```
(gdb) delete 3
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

end of function
account1 balance: $37

Program exited normally.
(gdb)
```

图 G.33　删除一个监视点

在这一节中，学习了如何使用watch命令使调试器能在程序的生命期中当数据成员的值发生变化时发出通知。同时，也学习了在程序结束前如何用 delete 命令从数据成员上删除监视点。

G.6　本附录小结

在本附录中，学习了如何在调试器中插入和删除断点。断点允许你暂停程序的执行，因此可以使用调试器的print命令检查变量的值。这种性能将有助于定位和修正存在于程序中的逻辑错误。我们看到了怎样使用 print 命令来检查表达式的值，学习了如何使用 set 命令来改变变量的值。同时，也学习了其他很多调试器命令（包括 step、finish 和 next 命令），它们能够用来判断函数是否正确执行。在一个数据成员的作用域内，可以用 watch 命令跟踪该数据成员的值的变化。最后，学习了如何使用 info break 命令列出为程序设置的所有断点和监视点，以及如何用 delete 命令来删除单个的断点和监视点。

摘要

- GNU包含了称为调试器的软件，它允许你监控程序的执行以便定位和消除程序中的逻辑错误。

[General Information]
书名=C++大学基础教程　（第5版）
作者=[美]H.M.DEITEL　DEITEL & ASSOCIATES,INC.　P.J.DEITEL著　张引等译
页数=650
SS号=11631746
出版日期=2006年06月

前言
正文

[General Information]
书名=C++大学基础教程　（第5版）
作者=[美]H.M.DEITEL　DEITEL & ASSOCIATES,INC.　P.J.DEITEL著　张引等译
页数=650
SS号=11631746
出版日期=2006年06月

前言
正文